异步图书
www.epubit.com.cn

/THEORY/IN/PRACTICE



# 视图更新与关系数据库理论

## 解决视图更新问题

View Updating & Relational Theory

[美] C. J. Date 著
田远帆 译

O'REILLY®
中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

# 视图更新与关系数据库理论

[美]　C.J.Date　　著

田远帆　译

人 民 邮 电 出 版 社

北 京

图书在版编目（ＣＩＰ）数据

视图更新与关系数据库理论 / (美) 达特
(C. J. Date) 著 ; 田远帆 译. -- 北京 : 人民邮电出
版社, 2017.2
ISBN 978-7-115-43546-0

Ⅰ. ①视… Ⅱ. ①达… ②田… Ⅲ. ①关系数据库系
统 Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字(2016)第272589号

## 版权声明



◆ 著　　　　[美] C.J. Date
译　　　　田远帆
责任编辑　陈冀康
执行编辑　胡俊英
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京昌平百善印刷厂印刷

◆ 开本：800×1000　1/16
印张：15.75
字数：292 千字　　　　2017 年 2 月第 1 版
印数：1 – 2 000 册　　　2017 年 2 月北京第 1 次印刷

著作权合同登记号　图字：01-2013-8796 号

定价：59.00 元

**读者服务热线：(010)81055410　印装质量热线：(010)81055316**
**反盗版热线：(010)81055315**

# 内容提要

本书是数据库专家 C.J. Date 的又一部力作，详细介绍了数据库架构的设计和实现，同时讨论了视图更新及关系数据库理论。

全书内容分为 15 章，用示例和理论相结合的方式，介绍了关系数据库的相关理论，还对视图作了深入探究。书中不仅介绍了一些基础的数据库操作，同时重点结合示例给出了视图相关的操作和讲解。通过阅读本书，读者能够在实际工作和学习中，改进自己的数据库设计，更好地实现数据库相关任务。

本书是经典的数据库参考指南，非常适合数据库架构师、数据库设计人员以及相关领域的研究人员参考阅读。

# 献辞

内涵的外延
Edgar F. Codd 发明了一种概念
就是我们知道的视图
现在的视图和基础关系变量
可相互转换
让我们高唱一曲
关于视图的布鲁斯

—Anon.:《Where Bugs Go》

“奥蒙德的公爵昨天审视了他的部队的全景，命令把所有的枣红色马和灰色马都换成黑马。”

——目前所知的最早关于视图更新的案例（1693），引自《Oxford English Dictionary》中的《A Brief Historical Relation of State Affairs 1678–1714》，作者 Narcissus Luttrell (1857)

浅尝辄止是如此危险；
诗泉之水若不深饮，弗如不尝：
浅浅一酌让人头脑发热，
尽兴狂饮却让人重新清醒。

——Alexander Pope《An Essay on Criticism》（1711）

—— ♦♦♦♦♦ ——

献给我最爱的妻子 Lindy、女儿 Sarah 和 Jennie

# 作者简介

C.J.Date 是一位专攻关系型数据库技术的独立作家、演说家、研究员和顾问。他最著名的著作《数据库系统导论》(第 8 版，Addison-Wesley，2004)销量已经突破 850000 册，并被全球数百所高校作为教材使用。他同时还撰写过许多其他数据库管理相关书籍，包括:

- Addison-Wesley 出版:《Databases, Types, and the Relational Model: The Third Manifesto》(第 3 版，与 Hugh Darwen 合著，2006)
- Trafford 出版: 《Logic and Databases: The Roots of Relational Theory》(2007)
- Apress 出版:《The Relational Database Dictionary, Extended Edition》(2008)
- Trafford 出版:《Database Explorations: Essays on The Third Manifesto and Related Topics》,(与 Hugh Darwen 合著，2010)
- Ventus 出版:《Go Faster! The TransRelational™ Approach to DBMS Implementation》(2002，2011)
- O'Reilly 出版:《SQL and Relational Theory: How to Write Accurate SQL Code》(第 2 版，2012)
- O'Reilly 出版:《Database Design and Relational Theory: Normal Forms and All That Jazz》(2012)

Date 先生于 2004 年正式进入计算机行业名人堂。他以能够用简明易懂的方式解释复杂的技术问题而著称，造诣之高，无出其右者。

# 前言

本书是这个系列的第 3 本书，它的两位“前辈”是：

- 《SQL and Relational Theory: How to Write Accurate SQL Code》（第 2 版）
- 《Database Design and Relational Theory: Normal Forms and All That Jazz》

以上两本书于 2012 年由 O’Reilly 出版发行。第 1 本书的目标读者是所有种类数据库的从业人员，书中解释了关系理论的基本原理，以及以此为基础如何将 SQL 当作一个关系型数据库使用(在那本书中我使用的一条准则是“关系化地使用 SQL”)。第 2 本书则针对性强一些，它瞄准的读者群是那些对数据库设计感兴趣的行业专家，书中解释了关系型数据库设计的一些理论，并展示了为什么这些理论如此重要。而第 3 本书，也就是本书，则针对性更强。本书专注于对一个非常关键的问题的研究，而这个问题则涉及关系型数据库应该如何运行（与目前大部分商业 SQL 数据库系统的运行表现恰恰相反）这一核心内容。这个问题就是“更新理论”。这个理论正如本书的题目显示的那样，适用于视图的更新——不论是在一般情况下，还是特定情况下。同时，本理论也适用于“基础数据”的更新。

注意：尽管我的理论包含上面的最后一句话，但我还是决定在本书的题目中更加强调对视图的更新，因为据我观察，一般数据库从业者相信他们自己能够理解对于基础数据为对象的更新是如何运作的，而一旦对象变成视图，他们的典型反应就是极度怀疑无论使用什么办法，对视图的更新能否真的实现。其实我一直很奇怪，关于视图更新的讨论居然曾经是并且依旧是一个有争议性的话题，当然这也是最初让我决定撰写这本书的一个重要原因。

顺带说一句，关于前两本书，我要对本书中大量引用它们（特别是第 1 本）而表示歉意。目前，本书中大部分引用的其他出版物都会给出全名，例如下面的例子。

David McGoveran：“Accessing and Updating Views and Relations in a Relational Database”美国，专利号 7263512（2007 年 8 月 28 日）

不过从现在开始，当我提到之前出版的著作时会使用简称，即《SQL and Relational Theory》和《Database Design and Relational Theory》。

作者注：我说过我会用全名引用其他的著作，不过其实也没有多少能引用的。虽然大量关于视图更新的论文、著作或者其他文章在最近 30 年内层出不穷，但它们中大多数，包括 David McGoveran 发表的某些例外的特定著作，所主张的论调和本书所体现的论点大相径庭（请参看前言后面部分关于这一点的讨论）。因此大多数时候，我都觉得不适合引用这些著作，除了偶尔因需要提及一下。如果你有兴趣详细地了解一下这些“其他的论调”，那么可以在《An Introduction to Database Systems》（第 8 版，Addison-Wesley，2004）的第 10 章中找到一个关于这些书籍名称的简短列表。

在这里我需要强调的是本书假定你们完全了解《SQL and Relational Theory》所讨论的内容。举例来说，我认为你们理应知道什么是关系，什么是属性，什么是数组，所以，我不会因为未对这些基础名词做提前解释而道歉。本书的目标读者是数据库相关专业人士，而专业人士应该对我之前的书所包含的这些基础知识很熟悉才对。为了使本书内容更加独立，我会在第 2 章“技术背景”中对前面的书的相关部分做一个简短的回顾。同时我也会在第 3 章“视图概念：近距离观察”中给出一个关于什么是视图以及它们应该如何运作的详细综述。

## 本书的目标读者

我的目标读者群是数据库相关专业人士，进一步说也可以是所有对关系模型、关系型技术或者关系型系统感兴趣的人。如前所述，对《SQL and Relational Theory》熟悉的话会对阅读本书有很大帮助，不过我相信本书无论对关系理论整体而言，还是对视图更新具体来讲，都会有新鲜的见解。而且，我认为值得指出的是，我很有可能可以利用这里面的一些想法在没有数据库管理系统（DBMS）[1]支持的情况下来指导一个“自助式”的实施（针对视图更新）。但在这里其实我真心希望数据库管理系统工程师们能够看到这本书，并由此能在他们自己的产品中提供对视图更新的支持。注意：我同时也想提一下，我会举行一场基于书中内容的现场研讨会。想了解更多内容，请访问下面网址：www.justsql.co.uk/chris_date/chris_date.htm。

## 本书的结构

如之前所说，我假设你知道什么是关系、属性、数组。更进一步来讲，我假设你也至少大体知道什么是视图。视图最初被讨论（当时并不用“视图”这个名字）是在 Codd 的第一篇关于关系模型的论文中：

[1] 数据库管理系统（DBMS）= database management system。很显然，数据库管理系统和数据库是有区别的！然而不幸的是，业界通常都把一些商业产品，如甲骨文，或者安装在特定电脑上的此类产品的特定版本称为“数据库”。在本书中我不会跟随这个潮流。问题在于，如果你把数据库管理系统称作“数据库”的话，那你要怎么称呼真正的数据库呢？

E.F.Codd: “Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks,” IBM 研究报告 RJ599（1969 年 8 月 19 日）

现在，正如 Codd 自己在这篇论文中所预见的，支持视图最主要的理由，是原则上可以通过它们来达成逻辑数据独立这个重要目标（逻辑数据独立指的是能够改变一个数据库的逻辑，而不需要相应改变用户的使用习惯的能力，由此可以保护针对既有用户培训、既有应用以及其他部分的投入。在第 3 章我们会有更深入的讨论）。换句话说，视图存在的主要理由，严格来讲就是为了实现逻辑数据独立这个目标。但如果我们想在实际中实现它，而不仅仅是停留在原则上，那么视图必须是可更新的。

所以针对视图的更新是一个很重要的问题。由此重要性导致的结果是，无论是对商业领域，还是研究领域而言，视图已经成为大家关注的焦点有一段时间了（至少有 35 年左右），而且一些不同的方案已经被提出，有些甚至被实施过。但可惜的是，所有这些方案都没能对视图更新给出一个令人满意的解决办法（这里我要强调，不仅仅是我，其他作者也这么认为）。我们举例来说，在当今主流的 SQL 产品中，视图更新机制具有以下两个典型的缺陷。

- 不完整，也就是说，它们没能使理论上应该能够更新的视图全都实现更新。
- 不正确，也就是说，即使对那些能够实现更新的视图，它们的实施方式也不对路，至少在某些情况下是这样的。

（再次说明，第 3 章中有关于这些问题的深度讨论。）至于那些研究文献，我觉得它们通常都会忽视一些重要因素，这些因素对于一个系统化的、综合性的以及正确的解决方案至关重要。与前面的例子不同，我相信在本书中将要详细讲解的，恰恰就是一个“系统化的、综合性的以及正确的” 解决方案。同时我也相信（在这里我必须说明，我本身并不是一名工程师），这个解决方案可以被整合进关系型数据库管理系统，并给系统架构带来适度的概念化扩展。

作者注：请注意我很谨慎地提到“关系型数据库管理系统”。在后面你将会看到，我提出的这个解决方案非常依赖对完整性约束的声明（当然，这需要数据库管理系统的强制执行能力来做保障）。从我的角度来说，我认为这些能力是构成一个真正的关系型系统的必要条件。不过相信大家都知道，当今绝大部分的 SQL 产品在这个环节都有缺陷。现在我正式介绍一下本书的大体结构。

- 第 1 章给出了一个抛砖引玉的例子，表明了在大家熟知的简单条件下（其实就是 SQL 的条件下）如何看待视图更新，在后面的章节中我们还会有更具体的

说明。具体来说，它证明了“更新就是更新”，无论被更新的是视图还是基础数据。这就是为什么我之前说这本书所涉及的问题应该被称为“更新理论”（一种既适用于视图，同时又适用于基础数据的理论）的原因。

- 接下来如前所述，第 2 章对关系理论的相关内容做了简要的回顾。其中特别强调了数据库的本质就是“一个真正的变量”，而它这个特质也恰恰让它成为对所有更新操作都适合的目标对象。
- 第 3 章对视图概念和相关内容做了更具体的描述。当然我前面说过我假定你了解视图的基本概念，但这一章包含了你可能不太熟悉的内容，而这些内容对于正确理解后面的内容至关重要。
- 第 4～13 章会针对各种各样你所熟悉（有的内容可能不那么熟悉）的关系操作逐个进行讨论——约束、投影、连接等。特别是在第 4 章，对于约束视图，我还介绍了很多额外的基础内容（其实，这几章也可以看作是第 3 章的一个继续）。第 4 章同时也给出了后面 9 章如何展开的计划。
- 第 14 章研究了合并操作（例如，更新一个由两个约束组成的连接，或者更新一个由两个连接组成的合并都需要什么）的问题，这个问题引发了一些相当有趣亦或出人意料的情况。
- 在整本书的讨论中，大家可能会对其中一些方案所提到的问题或内容不明确或产生混淆，最后在第 15 章我将为大家对这些不明确和混淆做出解释说明。
- 书后面还有两个附录。附录 A 的内容是关于所有重要的“关系赋值”操作的细节。附录 B 则收录了在本书主体中所使用或提到的大量关系操作的详细定义。

注意：目前给大家描述的大纲其实已经足以说明本书是按照循序渐进的方式来撰写的，我本人也强烈建议大家按照写作顺序来阅读本书。

## 技术性注释

还有几点我要在这里提前说明。首先请注意，在本书中我按照惯例使用一般术语，用小写字母表示的“更新”是代表包含“INSERT”“DELETE”以及“UPDATE”等所有操作语句的总称（正如我刚刚所提到的“所有重要的关系赋值操作”，详见第 2 章）。如果想表达“UPDATE”这个操作语句，我会全部使用大写字母表示，至于“INSERT”和“DELETE”操作就不会引起混淆。不过一味地使用大写会显得有些烦人，尤其是当它们作为修饰语使用的时候，例如“INSERT 规则”（“插入规则”？）。所以，我决定在本书中两种格式一起使用，我会根据前后语境自我发

挥（我也不会装作在这方面前后有多么一致）。

第二点，请注意我使用“SQL”来表示 SQL 语言的标准版，而不是什么专有的术语（另有明确说明的除外）。特别是按照标准读音“SQL”应该念为“ess cue ell”，而不是“sequel”（因为现在很多地方都按后者来发音），所以我对“一个 SQL 表”的书写会按照标准读音的写法。注意：SQL 标准版随着时间推移出现了数个版本，在本书写作的时候最新版本是 SQL：2011。以下是完整的版本信息。

国际标准化组织（ISO）：Database Language SQL, Document ISO/IEC 9075:2011 (2011)

第三点也是最后一点，我需要对我所谓的“用户”做一下说明，特别是我需要解释一下经常用到的一些短语，如“用户看到的”或者“用户对数据库的感觉”。总体来说，你可以把“用户”这个词理解为一个交互用户[1]、一个应用程序开发人员或者两者都是，请根据上下文理解。至于“用户看到的”和类似的短语中的用户，我指的是那些并未与整个数据库交互而更多的是与一些子集交互的用户，根据“子模式”定义。另外，根据视图的机制，子集可以并且经常参与到一些逻辑重建中。实际上，我们可以（至少我会这么做）简单而不失全面地假设，子集是由视图组成的，即使一些由基本数据转换而来的视图实际上跟导出它们的基本数据等价。当然，对于这个子集的用户来说，视图的集合就是数据库本身！换句话说，“数据库”在某种意义上来说是个相对项。所以，至少是为了本书使用，我们可以稍微有点随意但有效地把数据库定义为一个给定的数据，例如给定的基本数据的集合，或者是这个集合下的一些具体的，很可能是经过重构的子集。注意：当我在这里说到“随意”的时候，我考虑到的是这个数据库不仅仅只有数据，如相应的完整性约束也需要被考虑进去，我们在第 2 章和第 3 章将会见到。

[1] 这些用户依然是那些懂一点数据库问题的人，而不是那些纯的“最终用户”，他们可能对这些问题完全没有概念。

# 致谢

我想再次感谢我的夫人 Lindy 对我在撰写本书期间以及在撰写本系列前作时的大力支持。我还要感谢我的挚友和同事 Hugh Darwen、David Livingstone 以及 David McGoveran，他们对本书的早期草稿给予了大量细致的、综合的审阅。他们和他们所做的大量工作对本书都起到了重要作用，但我特别要感谢 David McGoveran。首先，感谢他最开始为本书的主题所提出的一些建议和想法，视图更新在本书中的描述大都基于此发展而来；其次，感谢他就这个主题在过去 20 年间跟我所做的沟通和协调；最后，感谢他在这个领域所进行的深远的理论工作。而且 David 所做的工作还大大超出了审阅本身，他不仅仅对词句给出建议，实际上他还针对本书主题的很多方面汇编并提供了许多短篇的论文或笔记。这些笔记在我重新修改文章的时候给了我非常大的帮助，我相信正是这些笔记让我的文章大有起色。当然，我并没有把他给的所有建议都采纳进来，我相信没有哪个作者完全按照审阅者的建议来行文！但是我已经尝试着尽可能采纳那些对我来说最重要和最真实的建议了。当然不用说，任何遗留的错误都是我自己的责任。

C.J.Date

于加利福尼亚，希尔兹堡

2013 年

# 序

在关系型数据库及其实际应用领域中，一直有两个特别棘手和富有争议的问题，而业界并没有得出一个大家都能满意的解决办法，这两个问题是：信息丢失问题和视图更新问题。对第 1 个问题，Chris Date 在过去 30 年左右的时间里已经撰写了大量著作，现在他开始着手应对第 2 个问题。

当然他在之前并非没有涉及视图更新这个问题。在他著名的著作，被作为教科书广泛使用的《An Introduction to Database Systems》中就有对该问题的讨论。在 1975 年第 1 版发行的时候，书中就有少量篇幅涉及这个问题；到第 8 版时（2004 年），书中已经有多达 16 页左右的篇幅来进行讨论。他第一次用整个章节讨论这个问题是在 1986 年开始撰写系列丛书《Relational Database Writings》的时候。到了 1995 年，在该系列第 4 本书中，他和 David McGoveran 带给了我们两个章节的内容来展示对这个问题思考的变迁，这些内容更多地源自 McGoveran 的研究。这个思考后来在《Databases, Types, and the Relational Model: The Third Manifesto》（2007）的附录中进化，在《Database Explorations》（2010）的章节中成长，最终到了今天的地位。

最初由 E. F. Codd 在 1969 年提出的基础理念，至今一直都没有变化。想象现在给定一个数据库，根据定义是由（a）可变关系或者“关系变量”[1]组成的集合，同时由（b）一套完整性约束管理着这些关系变量的可用值。这些给定的关系变量都是“基础”变量。总体来说，被选出的设计是在许多个设计中可以被选出来代表完全相同信息的那一个。从被选出的设计中我们可以根据基础关系变量的关系表达式，通过定义虚拟关系变量，从而得到一个备选方案。由于各种各样的原因，这样一个备选设计，实际上是一个备选的数据库视图，可能相对基础设计来说更适合某些特定用户。更重要的是，这个备选设计可能实际上并不包含底层的或者“真正的”数据库。有些用户对这些其实也不感兴趣，或者根本没有权限看到。除此之外，如果必须针对基础设计做出一些变化，代表原始设计的虚拟关系变量可以在新设计上被重新定义，这样一来已经存在的用户视图就无需改变，那些可能发生的令人讨厌的突变就可以避免。这就是著名的终极目标“逻辑数据独立”

[1] SQL 喜欢称这些关系变量为“表”。关于更多对于关系变量的名词解释和相关内容，请参看第 2 章。

背后的基础理念。

如果用户把虚拟变量视为他们的数据库，当实施与虚拟关系变量冲突的数据库更新时，棘手的问题就出现了。数据库管理系统到底是如何判定那些对真正的数据库实施真正的更新会对虚拟关系变量产生特定的改变的？如果这里有数种方式可以实现需要的效果，那么到底该选用哪种方式呢？举个简单的例子，假设一个用户使用常见的“供应商与零部件”数据库（第 1 章有详细介绍），他看到一个虚拟关系变量或者视图命名为“PS”，其中只显示位于巴黎的供应商。毋庸置疑，PS 的定义表达式是“S WHERE CITY = 'Paris'”。现在，我们假设这个用户让数据库管理系统从视图 PS 中删除数组 S2。那么数据库管理系统是否应当这样理解用户的意图：供应商 S2 不存在了，需要从基础关系变量 S 中删除相应的底层数组？还是说应当因为用户表述不清而拒绝客户的删除指令，因为将供应商 S2 的 CITY 值换成巴黎以外的地方也会达成相同的效果？或者在另一种情况下，假设用户知道供应商 S2 已经搬迁到伦敦，因此尝试在视图 PS 中对 S2 进行“更新数组”的操作来达到从视图 PS 中删除数组 S2 的效果，那么数据库管理系统是否应当接受这个更新操作呢？现在我们进一步假设视图 PS 没有 STATUS（状态）属性。那么数据库管理系统应当如何回应该用户对视图尝试插入数组的操作，而用户要求这些数组必须要删除 STATUS 值？

针对以上和更多的此类问题，Date 都试图在他详细的、全面的、细心的、有条理的分析中为我们做出解答。他计划在最开始的 3 个章节中对这些问题发起进攻。他清楚地定义了“2 个数据库被设计为在表达相同信息的时候是等价的”是什么意思，然后在接下来的 10 个章节，他将对他所使用的方法进行详细讲述。这个方法需要轮流检验每一个关系代数中的操作。例如，那个“只包含巴黎供应商”的视图 PS 被他称为约束视图，比如，定义一个虚拟关系变量只使用约束操作。同样地，现在定义一个不包含 PS 中 STATUS 属性的视图使用投影。既然这个视图是一个约束的投影，我们可以来推断它的更新效果。首先根据 Date 的更新规则通过投影来决定对底层约束的效果，然后根据更新规则通过约束来决定对底层基础关系变量 S 的效果。

对由一些关系操作定义的视图应用这些规则的时候会产生一个非常有趣，甚至可以说是很有争议性的问题，这个问题 Date 在第 14 章会做阐述，那就是如果两个表达式在语法上不同，但逻辑上等价。例如，数学表达式 $x(y+z)$ 和 $xy+xz$ 在语法上不同，但逻辑上是等价的。那么定义于它们之上的视图在执行更新操作的时候运行行为（如过程、动作或效果等）是否一定要完全相同？

现在，Date 的一些观点在我前面提到的 2007 年和 2010 年的出版物上出现的时候，

被证明是具有争议性的。例如，当我们向 R1 和 R2 的合并视图插入 1 个数组时，这个数组是否应该同时出现在 R1 和 R2 之内？如果我们从 R1 和 R2 的交集视图中删除一个数组时，是否会导致这个数组从 R1 和 R2 中同时消失呢？作为针对那些具体观点表达不同意见的一员，在这里我要强调，在我和 Date 长时间的合作过程中，我们之间唯一非常严重的技术分歧已经出现。这些争议点在本书中也有展现，并且 Date 已经针对这些问题加强了基本原理的解释，不过这可能仍然无法让所有人都信服他的观点。而我发现，在他的最后一章“歧义问题再回顾”中出现了希望之光，就好像是隧道尽头的那一盏明灯。在其中他描述了一个想法的大概轮廓，由 David McGoveran 提出的一种与以往我们更新关系型数据库所采用的方式完全不同的方式，它可以很有效地替代，或者至少是拓展，我们所熟知的“插入”“删除”和“更新”这些早在关系型时代之前就与我们紧密相连的操作。而这种不同寻常的方式所带来的好处是，我之前提到的会产生争议的问题根本不会出现了。

Date 告诉我，他并没有期待，甚至没有想过让本书成为他关于视图更新故事的结尾，但他希望本书能够给现存的争论提供一个坚实的理论基础，以使得整件事情可以向前推进。我想这正是他在本书中所提供的，我在这里也希望本书对你有相同的效果。

Hugh Darwen

于英格兰 Shrewley 区

2013 年

# 目录

第 1 章

# 抛砖引玉

身教胜于言教。

——Samuel Johnson《Rasselas》（1759）

本书中所涉及的大多数范例都是基于我们耳熟能详的（可别说是老掉牙的）“供应商与零部件”数据库而来的。我为再一次把这个老生常谈的例子拿出来表示歉意，但正如我在其他地方所说，我认为在各种不同的出版物上使用同样的例子对于学习而言是有利无害的。就SQL来说[1]，数据库包含 3 张表，更确切地说，是 3 张基表，分别称为供应商表S（“suppliers”）、零部件表P（“parts”）和设备供应表SP（“shipments”）。样 本值如图 1.1 所示。

这几张表简要的语义如下。

- 表 S 的内容列出了签过合同的供应商。每个供应商有一个唯一的供应商编号 SNO、可能不唯一的供应商名字 SNAME（尽管图 1.1 中的样本值恰好是唯一的），以及状态值 STATUS 和位置 CITY。注意：在本书的其余部分，我将把“签过合同的供应商”简称为“供应商”。
- 表 P 的内容表示各种各样的零部件。每一种零件都有唯一的零件编号 PNO、名称 PNAME、重量 WEIGHT 和用于存储同种类零部件的位置 CITY。注意：在本书其余部分，我将把“各种各样的零部件”简称为“零部件”。

[1] 我在本章这个介绍性的章节使用 SQL 和 SQL 风格的语法是因为大家都对它比较熟悉，尽管它并不是我喜欢的风格，并且（更重要的是）事实上它可能会让这个抛砖引玉的例子更不容易解释清楚。

- 表 SP 的内容表明了供应情况，即显示了哪种零件被哪一家供应商运输或供应。每一次运输都有供应商编号 SNO、零件编号 PNO 以及数量 QTY。同时，对于特定的供应商和零件，在某一时期，最多只能有一次运输。并且像这样的供应商编号和零件编号组合对任意一次运输都是唯一的。注意：本书的其余部分，我将假设 QTY 值总是大于零的。

S

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

P

| PNO | PNAME | COLOR | WEIGHT | CITY |
|---|---|---|---|---|
| P1 | Nut | Red | 12.0 | London |
| P2 | Bolt | Green | 17.0 | Paris |
| P3 | Screw | Blue | 17.0 | Oslo |
| P4 | Screw | Red | 14.0 | London |
| P5 | Cam | Blue | 12.0 | Paris |
| P6 | Cog | Red | 19.0 | London |

SP

| SNO | PNO | QTY |
|---|---|---|
| S1 | P1 | 300 |
| S1 | P2 | 200 |
| S1 | P3 | 400 |
| S1 | P4 | 200 |
| S1 | P5 | 100 |
| S1 | P6 | 100 |
| S2 | P1 | 300 |
| S2 | P2 | 400 |
| S3 | P2 | 200 |
| S4 | P2 | 200 |
| S4 | P4 | 300 |
| S4 | P5 | 400 |

图 1.1　供应商与零部件数据库的样本值

现在让我们把重心放在表 S 上。在本章的其余部分，我们基本上不再关注表 P 和表 SP，只在个别地方提及。以下是表 S 的 SQL 定义。

```
CREATE TABLE S
  ( SNO VARCHAR(5)    NOT NULL ,
    SNAME VARCHAR(25) NOT NULL ,
    STATUS INTEGER    NOT NULL ,
    CITY VARCHAR(20)  NOT NULL ,
    UNIQUE ( SNO ) ) ;
```

如前所述，虽然表 S 是基表，但我们仍然可以定义任意数量的“基于”这个基表的视图。这里有两个例子——伦敦供应商 LS（“London suppliers”）和非伦敦供应商 NLS（“non London suppliers”）。

```
CREATE VIEW LS /* London suppliers */ AS
  ( SELECT SNO , SNAME , STATUS , CITY
    FROM S
    WHERE CITY = ‘London’ ) ;

CREATE VIEW NLS /* non London suppliers */ AS
  ( SELECT SNO , SNAME , STATUS , CITY
```

```
       FROM S
       WHERE CITY <> 'London' ) ;
```

这两个视图的样本值与图 1.1 基表的对应关系如图 1.2 所示。

LS

| SNO | SNAME | STATUS | CITY |
| --- | --- | --- | --- |
| S1 | Smith | 20 | London |
| S4 | Clark | 20 | London |

NLS

| SNO | SNAME | STATUS | CITY |
| --- | --- | --- | --- |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S5 | Adams | 30 | Athens |

图 1.2　视图 LS 和 NLS 的样本值

我将在本章中以视图 LS 和 NLS 为基础来阐述整个范例，以求抛砖引玉。实际上，我相信视图都是可更新的，这似乎与本领域内的流行说法和最常见的认知格格不入，但在这里我将向大家展示一些关于这个问题基本的理念，以及为什么我会这样想。（不过请注意：在现阶段我只会对这个吸引眼球的想法做必要的说明，详细的探讨我们放到后面的章节再来进行。）

## 1.1　可交换性原则

到目前为止，表 S 是基表，LS 和 NLS 是视图。现在仔细观察，你会发现其实我们还可以用另一种方式来看它们，那就是把 LS 和 NLS 作为基表，而 S 作为视图，具体如下所示。

```
   CREATE TABLE LS
      ( SNO    VARCHAR(5)  NOT NULL ,
        SNAME  VARCHAR(25) NOT NULL ,
        STATUS INTEGER NOT NULL ,
        CITY   VARCHAR(20) NOT NULL ,
        UNIQUE ( SNO ) ) ;

   CREATE TABLE NLS
      ( SNO VARCHAR(5) NOT NULL ,
        SNAME VARCHAR(25) NOT NULL ,
        STATUS INTEGER NOT NULL ,
        CITY VARCHAR(20) NOT NULL ,
        UNIQUE ( SNO ) ) ;

   CREATE VIEW S AS
```

```
( SELECT SNO , SNAME , STATUS , CITY
  FROM LS
  UNION
  SELECT SNO , SNAME , STATUS , CITY
  FROM NLS ) ;
```

为了保证现在这个设计与原先的设计完全等价，我应该声明并且让数据库满足特定的完整性约束，当然其中应该包括针对 CITY 值的一致性约束，以保证 LS 中每一个 CITY 值都是 London，NLS 中 CITY 值都不是 London。但是现在我们暂时不考虑这些细节，稍后我会对这些问题做更详细的解释。

那么现在，做出上面这个变化所要表明的内容可以概括为：对基表和视图的设定是没有限制的，是可以互换的（至少从正式的角度来看是这样）。换句话说，对于这个例子，我们至少可以用两种方式设计数据库，这两种方式逻辑上不同，但包含的信息是等价的。（这里的等价信息，是指两个设计代表了相同的信息，也就是说对于在一种设计执行的查询，一定有在逻辑上和它等价的查询可以在另一种设计上执行。第 3 章将详细阐述这个概念。）而“可交换性原则”就是对上面内容的总结。

- **定义：**可交换性原则说明在基表和视图之间一定不能存在不确定的和不必要的区别。换句话说，在用户看来，视图与基表应该看起来感觉是很像的。

关于这个原则，下面几点值得注意。

- 正如我所阐述的，视图与基表一样，受完整性约束的限制（我们通常认为完整性约束是专门应用于基表的，但是可交换性原则证明这种想法有失偏颇）。
- 特别要说的是，视图是有键的（我确实应该在视图的定义中包括键的声明。但可惜的是，SQL并不允许这样声明）[2]。同时，视图也可能有外键，而外键也可能引用自视图。
- 很多 SQL 产品和 SQL 标准，提供了某种叫作 row ID 的特性（在标准中，此特性被称为 REF 类型和引用值）。如果这个特性只对基表有效，而对视图无效（实际上，这种情况非常有可能发生），那就明显违反了可交换性原则。
- 最后，也许最重要的是，我们必须能够更新视图，因为如果不能实现更新，那

[2] 在本书中我都使用未经认证的“键”这个词来表示候选键，而不是一个特定的主键。实际上，在第 2 章中就会介绍到的“Tutorial D”对于主键和其他键在语法上并没有区别。但是，由于大家使用习惯的原因，我在图例中都使用双下画线来表示主键属性，如图 1.1 所示。

就等于“可交换性原则”内部自相矛盾了。

## 1.2 仅限基表：约束

另一个由可交换性原则引出的问题是表 S、表 LS 和表 NLS 的行为不应该由基表和视图的分布来决定。为了解释这个问题，我们先假设它们全都是基表。

```
CREATE TABLE S   ( ... , UNIQUE ( SNO ) ) ;
CREATE TABLE LS  ( ... , UNIQUE ( SNO ) ) ;
CREATE TABLE NLS ( ... , UNIQUE ( SNO ) ) ;
```

这样一来，所有的表就都受到约束的限制了。不过从实际情况来看，想在 SQL 中构造这些约束条件会非常复杂，因此为了便于表达和理解，我将用普通话来描述一下这些约束条件（大部分是大白话）。

- {SNO}是每个表的键，同时，{SNO}对于LS和NLS来说是外键，引自于表S。注意：关于使用“{”和“}”的原因，请参阅《SQL and Relational Theory》[3]。

- 在任何时候，表LS都等同于表S中CITY值为London的限定，表NLS都等同于表S中CITY值不为London的限定。同时，表LS中每一行的CITY值都为London[4]，表NLS中的每一行都不为London。

- 在任何时候，表 S 都等于表 LS 和 NLS 合并后的结果。同时要注意，这个合并是不相交的（对应的交集为空），没有数据会同时出现在表 LS 和表 NLS 中。具体来说就是：表S中的每一行都会根据限定条件准确地出现在表LS或表NLS中，而表 LS 或表 NLS 中的每一行也必定会出现在表 S 中。

- 最后，刚刚所说的由 CITY 值引发的约束以及{SNO}是所有三张表的键这个约束结合在一起，就会形成以下结果，即表 S 中每个具体的供应商编码（不仅是整行的信息）都会根据限定条件准确地出现在表 LS 和表 NLS 当中。而表 LS 或表 NLS 中的每一个具体的供应商编码也必定会出现在表 S 中。

当然，正如前面所说的几点所示，这些约束并不是完全独立毫不相干的，其中一些约束是另一些约束逻辑上导致的结果。

---

[3] 我在前言的时候就声明过，在本书中我将使用《SQL and Relational Theory》作为简写来引用我的另一本书《SQL and Relational Theory: How to Write Accurate SQL Code》（第 2 版，奥莱利出版，2012）。

[4] 正是由于这个原因，更真实一点版本的视图 LS 很可能会去掉 CITY 属性。在这里我选择不这样做，是为了让这个例子更简单易懂。

# 1.3 仅限基表：补偿性操作

现在，为了确保之前章节提到的约束在后续的某些特定更新升级完成后依然有效，就需要进行相应的**补偿性操作**。通常情况下，补偿性操作也称为补偿操作，是一种由数据库管理系统自动执行的额外更新（不包括用户自行请求的更新），其目的就是为了避免完整性冲突的发生[5]。级联删除就是一个典型的例子[6]。事实上，有一点我们要明确，对于现在手上这个例子来说，我们很需要在进行DELETE操作时应用级联方式。具体来说，不管是从表LS，还是表NLS中，删除数据都需要级联以使得表S中相同的数据行也能够被删除。于是我们可以设想一些补偿性操作，其实就是级联删除规则，观察以下伪代码。

```
ON DELETE d FROM LS : DELETE d FROM S ;

ON DELETE d FROM NLS : DELETE d FROM S ;
```

同样，从表 S 中删除行也需要级联以使表 LS 或表 NLS 中同样的行一并被删除。

```
ON DELETE d FROM S : DELETE ( d WHERE CITY = 'London' ) FROM LS ,
                     DELETE ( d WHERE CITY <> 'London' ) FROM NLS ;
```

这里我要特别提醒的是，任何想对并不存在的行进行删除的尝试都是无效的，因此我们可以将圆括号内的表达式直接替换为 d。不过从定义方式的角度来讲，圆括号内的定义表达方式更好，因为它们对变量的定义更加具体明确。

类似地，我们也需要为 INSERT 操作设定补偿性操作，即级联插入规则。

```
ON INSERT i INTO LS : INSERT i INTO S ;
ON INSERT i INTO NLS : INSERT i INTO S ;
ON INSERT i INTO S : INSERT ( i WHERE CITY = 'London' ) INTO LS ,
                     INSERT ( i WHERE CITY <> 'London' ) INTO NLS ;
```

当然，级联插入的概念并不经常与外键约束联系在一起，但这个概念并没有问题。重要的是我们要认清补偿性操作并不总是以简单级联的形式出现。我们在本章中所讨论的仅仅是个简单的例子，以便大家理解概念，在实际工作中则需要根据具体情况来应用各种更复杂的形式，在后面的章节我们也会有一些案例给大家展示。

---

[5] 曾经有一位审阅者问我，为什么我在这里要选择"补偿性操作"这个词。当时我自己认为这个答案很明显，不过为了以防万一，我还是在这里做一个说明：之所以我把这些操作称为"补偿性"的，是因为它们都会导致第 2 次更新发生来补偿第 1 次的效果（当然是大体上来讲）。

[6] 级联删除通常都被认为是使用一个特定的外键约束，但是补偿性操作的概念其实更加通用，并且适合很多种约束。

在当前的范例中，UPDATE 操作可以被看作 DELETE 和 INSERT 的结合。因此，简单地说，对本范例的必要的补偿性操作就是对 DELETE 和 INSERT 操作规则设定的总和。例如，考虑下面表 S 上面的 UPDATE 操作。

```
UPDATE S
SET    CITY = 'Oslo'
WHERE  SNO = 'S1' ;
```

产生的结果如下。

1．表 S 中供应商 S1 现有的行被删除，CITY 值为 Oslo 的新行被插入到这个表中。

2．表 LS 中供应商 S1 现有的行同样被删除，这是因为从表 S 到表 LS 的级联删除规则。同时因为 CITY 值为 OSlo，这个供应商的新行被插入到表 NLS，这次则是应用了级联插入规则。换句话说，供应商 S1 的行从表 LS 迁移到了表 NLS（当然，这只是笼统来看）。

现在我们假设原来的 UPDATE 操作是在表 LS 中执行，而不是在表 S 中执行的。

```
UPDATE LS
SET    CITY = 'Oslo'
WHERE  SNO = 'S1' ;
```

产生的结果如下。

1．供应商 S1 现有的行从表 LS 中删除。

2．尝试向表 LS 插入供应商为 S1 的新行，CITY 值为 Oslo。但是该尝试会失败，因为它违反了表 LS 中 CITY 值必须为 London 的约束。因此这次 UPDATE 操作整体失败，之前已经完成的步骤（在表 LS 中删除供应商 S1 的行）被撤销，最后的结果就是数据库保持不变。

## 1.4　视图：约束和补偿性操作

现在我要开始讨论本章的核心概念。如果前面两段中所涉及的表中有一部分或者全部都是视图的话，那么我们讨论的所有结论依旧成立，没有改变。例如，像之前一样，假设 S 是表，LS 和 NLS 是视图。

```
CREATE TABLE S      ( ............. , UNIQUE ( SNO ) ) ;
CREATE VIEW  LS  AS ( SELECT ... WHERE CITY = 'London' ) ;
CREATE VIEW  NLS AS ( SELECT ... WHERE CITY <> 'London' ) ;
```

现在假设用户只能看到视图 LS 和 NLS，但是希望像基表一样操作它们。在这个用

户看来，这些表的语义如下。

LS：供应商 SNO 是已经签约的，名称为 SNAME，有状态值 STATUS，位于城市 CITY 中（London）。

NLS：供应商 SNO 是已经签约的，名称为 SNAME，有状态值 STATUS，位于城市 CITY 中（非 London）。

此用户将会了解有下面的约束存在（注意这些约束没有提到表 S，因为用户并不知道表 S 的存在）。

- {SNO}是 LS 和 NLS 的键。
- LS 中的每一行 CITY 值均为 London，NLS 没有这样的行。
- 供应商编号不会同时出现在 LS 和 NLS 中。

但是，该用户并不会意识到有补偿性操作的存在，因为他并不知道 LS 和 NLS 实际上是表 S 的视图。如前所述，该用户甚至根本不知道表 S 的存在，因此他也并不知道实现这些操作的约束到底是什么，以及视图 LS 和 NLS 合起来就等于表 S 这个事实。该用户执行的针对视图 LS 和 NLS 的更新也都会生效，在他看来就像 LS 和 NLS 都是基表一样。同样，这个用户在视图 LS 和 NLS 执行的更新在 S 中也会产生对应的关联效果，只是他看不到而已。

## 1.5 规则至上

现在我们假设用户只能看到视图 LS（而不是视图 NLS 和基表 S）。可以想见这个用户仍然希望可以将视图 LS 当作基表一样操作。当然，这个用户知道表的语义。

LS：供应商 SNO 是已经签约的，名称为 SNAME，状态值为 STATUS 且位于城市 CITY 中（London）。

并且了解如下约束。

- {SNO}是 LS 的键。
- LS 中的每一行 CITY 值为 London。

很明显，这个用户无法向这个表插入新行，也不能更新表中的供应商编码，因为这样的操作很可能违反约束规则，而用户并不知道这些约束（必须不知道）[7]。但是

[7] 我假设有些用户“可能”会被允许进行这样的操作，如果他或她在操作被拒绝的时候能够接受错误信息显示的只是“因为系统就是这么说的”，而没有更进一步的解释。关于这一点的进一步讨论详见第 4 章。

如果LS是一个基表，那么用户完全可以插入新行。否则，这张表将永远是空的！那么，前面所说的事情不就违背“可交换性原则”了吗？

事实并非如此。尽管这个特定的用户不被允许向这个表插入新行，但并不是所有用户都不可以。这个用户不能插入新行到 LS 的基本原因是他只是管中窥豹，无法看到全局。对比来看，我们在上一部分所讲的用户可以同时看到 LS 和 NLS，它们合起来等价于表 S，因此该用户相当于能够看到全局，也由此他就可以向 LS（和/或是 NLS）插入新行。但是只能看到 LS 的用户等于只能看到一部分信息，所以一些特定操作是不允许他执行的。

最后，值得指出的是即便我们遇到的数据格式都是基表，用户也会遇到类似的情况。也就是说，即使涉及的表都是基表，有时某些用户也会被禁止在某些特定的表上执行某些特定的更新。举个例子，让我们想想只能看到基表 SP，但是看不到基表 S 的用户。他会和只能看到表 LS 的用户一样，无法执行插入操作，因为这样的操作同样可能违反用户并不知道的（必须不知道）约束规则，这个约束就是从 SP 到表 S 和 P 的外键约束。

## 1.6 小结

抛砖引玉到此就要告一段落了。我们所用到的这个例子非常简单，从中得出的结论也是显而易见的。而其实我是想通过这个例子引出下面的理念，就是根据实际的定义把视图看作与特定的表相附生的基表，这对于整体考虑如何解决视图更新的问题是很有建设性的。我认为它不仅具有建设性，而且是一种“逻辑上正确”的方法。[8]总体思路如下。

1．视图定义表达式包含了某些特定的约束。例如，视图 LS（“London suppliers”）的视图定义表达式表明了 LS 等同于表 S 中 CITY 值为 London 的约束。

2．这种约束反过来表明了特定的补偿性操作（如那些为了避免违反完整性约束，而需要在用户自行请求的更新之外进行的操作）。举例来说，正如前面讲到的，表 S、LS 和 NLS 的约束就代表相应的级联删除和级联插入。

这里我要强调第 2 点——这点很重要，就是在某些状况下，补偿性操作可以由相关视图的定义表达式决定。换句话说，这样的操作不需要显式指定，也不需要给已经超负荷的DBA[9]增加工作负担。对于这个问题和其他在本章中引出的问题，我都会在后面的章节进行更详细的探讨。

---

[8] 在此感谢 David McGoveran，他在多年前启发我思考这些问题。

[9] DBA=database administrator，也就是数据库管理员。

最后，如果你像大多数人一样跳过了前言直接开始阅读第 1 章，那么现在是时候翻回去阅读前言了，这对你继续阅读下面的章节会很有帮助。前言包含了本书的整体结构，同时还阐明了将要在下面的章节使用的重要的技术性假设，因此需要你对它们有所了解。

# 第 2 章

# 技术背景

适合我的一切也应该能适合你。

——Walt Whitman《Leaves of Grass》（1885）

上一章的讨论是基于SQL的，因为大家对它都很熟悉。但实际上很可惜，SQL并不适合作为这种探究的基础，也无法满足目前手上课题对细节技术讨论的要求。一方面来讲，我们需要检验的概念很多时候完全无法用SQL语句表达；从另一方面来看，即使是可以表达的时候，SQL通常也会引入一大堆与之毫无关系而又没有必要的复杂内容，很容易使人一叶障目，不见森林。由于以上原因，在本文余下部分中，我将不会使用SQL作为论述基础（不过有些地方我还是要针对SQL做出一些说明），而是使用一种假想的语言，名字叫Tutorial D[1]。我相信现在这个语言已经广为人知了。不过，如果你需要一个比较综合的解释说明的话，可以在我和Hugh Darwen合著的《Databases, Types, and the Relational Model: The Third Manifesto》（第 3 版，Addison-Wesley出版，2006）[2]这本书中找到。

正如上面这本书的标题所写，后文中我会用“关于宣言的那本书”作为简称来引用，它还为大家介绍和解释了“第三宣言”，这是一个很精准的对于关系模型和支持类型理论（也包含了一个关于类型继承的复杂模型）的正式定义。在那本书中，我们

[1] 说这个语言是假设的，仅仅是因为到我撰写本书的时间为止，市场上还没有一个正式发售的商业版本。但是它的设计原型已经存在，可以通过下面的网址来访问：www.thethirdmanifesto.com （请参看下面紧跟着的另一个注脚）。

[2] Tutorial D 从该书第 1 次出版以来已经被修订和扩展。关于新版本（已经有改善，但和本书中使用的并不完全相同）的描述可以在我和 Hugh Darwen 合著的《Database Explorations: Essays on The Third Manifesto and Related Topics》（Trafford, 2010）中找 到，同样可以访问网站 www.thethirdmanifesto.com 来获得相关资讯（这本书同时还有很多其他关于“宣言”的内容）。

使用“D”作为所有能够符合“宣言”原则要求语言的通用名。许多优秀的语言都够得上“D”的资格，很可惜，SQL 是个例外。反观 Tutorial D 就是一个合格的“D”。实际上，Tutorial D 原本就是被设计为一个说明和讲解“宣言”理念的合适载体，它也同样适合在本书中作为我研究的基础语言。因此，虽然我的说法是本书中的讨论要以 Tutorial D 为基础，而其实更准确的说法应该为这些讨论本质上是建立在“宣言”的理念基础上的。本章的其余部分包含一个关于这些理念想法的调查（当然，调查并不包含与我们的文章主题关系不大的那些观点）。换句话说，它主要是由一些你可能已经比较熟悉的内容组成的。即使如此，这一章仍然值得你至少“从头到尾马马虎虎地”读一遍，只要能对后面章节所倚重的概念和名词做些了解就可以。

## 2.1 关系和关系变量

每一个关系都有一个“关系头”和一个“关系体”，其中关系头是一系列状态和特征，而关系体则是符合关系头的一系列数组。让我们再次使用“供应商与零部件”数据库（如图 2.1 所示，与第 1 章的图 1.1 完全一样），供应商关系的关系头是{SNO CHAR, SNAME CHAR, STATUS INTEGER, CITY CHAR}的集合，我们假设在这里确定属性 SNO、SNAME、STATUS 和 CITY 分别被定义为 CHAR、CHAR、INTEGER 和 CHAR 这几种数据类型，而这个关系的关系体是对于供应商 S1、S2、S3、S4 和 S5 的数组的集合。这里要注意，每个数组都符合特定对应的关系头的要求，因为它们每个都只包含一个值，分别对应属性 SNO、SNAME、STATUS 和 CITY（并且没有其他内容）。注意：其实更准确，也是更正确的说法应该是每个数组都“拥有”特定对应的关系头，因为在一个关系中，数组是应当有关系头的。

S

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

P

| PNO | PNAME | COLOR | WEIGHT | CITY |
|---|---|---|---|---|
| P1 | Nut | Red | 12.0 | London |
| P2 | Bolt | Green | 17.0 | Paris |
| P3 | Screw | Blue | 17.0 | Oslo |
| P4 | Screw | Red | 14.0 | London |
| P5 | Cam | Blue | 12.0 | Paris |
| P6 | Cog | Red | 19.0 | London |

SP

| SNO | PNO | QTY |
|---|---|---|
| S1 | P1 | 300 |
| S1 | P2 | 200 |
| S1 | P3 | 400 |
| S1 | P4 | 200 |
| S1 | P5 | 100 |
| S1 | P6 | 100 |
| S2 | P1 | 300 |
| S2 | P2 | 400 |
| S3 | P2 | 200 |
| S4 | P2 | 200 |
| S4 | P4 | 300 |
| S4 | P5 | 400 |

图 2.1 供应商与零部件数据库的样本值

每个关系也都拥有（或者说属于）一个特定的“类型”，而这个特定的类型其实完全是由特定的关系头决定的。因此，我们在 Tutorial D 中表示一个给定的关系的类型时，其实就是关键词“RELATION”后面跟着适用的关系头。举例来说，供应商关系的类型就是：

```
RELATION { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
```

接下来我要说，关系本身和关系变量之间存在着逻辑差异 [3]。现在我们再来看一下图 2.1。这张图呈现了 3 个关系，也就是说，这 3 个关系有可能在特定的时间里同时存在于数据库中。但是，如果我们换个时间再来看数据库的话，很有可能我们看到的就是其他的关系。换句话说，S、P和SP都是变量，准确地说是关系变量，它们跟其他变量一样，值会随时变化。那么既然它们是具体的关系变量，那么在给定的时间内，它们的值就是关系值。不过要注意，对于一个给定的关系变量来说，合法的值必须都属于同样的关系类型，例如，这些值必须都有相同的关系头，而这个关系类型和关系头也因此被看作这个关系变量的类型和头。

为了更全面地阐述前面提到的想法和理念，我们假设关系变量 S 拥有和图 2.1 所示相同的值，我们再假设现在要删除在伦敦的供应商的数组。

```
DELETE ( S WHERE CITY = ‘London’ ) FROM S ;
```

关系变量 S 现在应改变成如下的样子。

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S5 | Adams | 30 | Athens |

从概念上来讲，S原来的值（当然是个关系值）已经被一个新的值（另一个关系值）整体替换掉了。现在我们来看，原来的值（有 5 个数组）和新的值（有 3 个数组）似乎看起来很像，但是其实它们是完全不同的值。我们用“DELETE”作为刚才操作语句的缩写 [4]，实际上在这里它等价于下面的“关系赋值”。

[3] “逻辑差异”这个概念，是我和 Darwen 在撰写技术文章时很爱使用的一个概念（特别是在关于《The Third Manifesto》的文章中），它来源于 Wittgenstein（英国哲学家、数理逻辑学家维特根斯坦）的一句名言：All logical differences are big differences（所有逻辑上的差异都是巨大差异）。

[4] “缩写”在这里用引号括起来是因为其实它要比需要缩写的部分还长。但是一部分原因其实应该归咎于我使用的是不规范的 Tutorial D 术语（在真正的 Tutorial D 语法中，DELETE 语句应该写成 DELETE S WHERE CITY = ‘London’）。在本书中我特意选择了稍微啰嗦一点的语法来表达 DELETE（和 INSERT），这也是为了让它们的语义足够清晰。

```
S := S WHERE NOT ( CITY = 'London' ) ;
```

或者等价于：

```
S := S MINUS ( S WHERE CITY = 'London' ) ;
```

那么经过所有这些赋值操作会发生什么呢？我们会看到（a）在右侧的源表达式是需要运算的，而（b）运算得出的值就会被赋给等式左侧的目标变量，于是就会产生我们刚刚讲过的效果。

因此 DELETE 是一个特定关系赋值的缩写，当然，对于 INSERT 和 UPDATE 来说也是相似的，它们从本质上来说也是对于特定的关系赋值的一个缩写。从逻辑上来讲，实际上关系赋值是我们唯一真正需要的更新操作（这是我在下一节要重点说明的观点）。

综上所述：关系本身和关系变量之间存在着逻辑差异。也正因为如此，我在之后也会很谨慎地区分它们二者，当我用"关系值"这个词的时候我就是在指关系的值，而当我说"关系变量"这个词的时候指的就是关系的变量。不过，我大多数时间还是会用"关系"作为"关系值"的简称（就好像我们用"整数"来作为"整数值"的缩写一样）。并且我大多数时间会用"**relvar**（关系变量，译者注）"来作为"关系变量"的缩写，例如，我会说"供应商与零部件"数据库包含 3 个"关系变量"（更具体一点来说，是 3 个"真实"或基础关系变量，之所以这么说是为了把它们和"虚拟"关系变量或者视图区分开）。

### 基础关系变量定义

下面是 Tutorial D 对于目前例子中使用的 3 个关系变量的定义，以供后面参考引用。

```
VAR S BASE RELATION
  { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
  KEY { SNO } ;

VAR P BASE RELATION
  { PNO CHAR , PNAME CHAR , COLOR CHAR , WEIGHT RATIONAL , CITY CHAR }
  KEY { PNO } ;

VAR SP BASE RELATION
  { SNO CHAR , PNO CHAR , QTY INTEGER }
  KEY { SNO , PNO }
  FOREIGN KEY { SNO } REFERENCES S
  FOREIGN KEY { PNO } REFERENCES P ;
```

出于本书的目的，我选择忽略 Tutorial D 在目前定义中并不包含明确的 FOREIGN KEY 语法的这个事实。

## 2.2 关系赋值

Tutorial D 中的关系赋值语法，不是像 INSERT 或 DELETE 一样的缩写，而是使用如下的通用格式。

```
R := rx
```

在这里“R”是一个关系变量（从语法上来说，其实就是一个关系变量的名字），“rx”是一个关系表达式，表示关系“r”和关系变量“R”是同一类型的。现在显而易见的是（这里要感谢 David McGoveran 的观察）任何这类赋值都在逻辑上等价于下列格式之一。

```
R := ( r MINUS d ) UNION i
```

在这里：

- r 是 R 的“原有”值。
- d 是将要从 R 中删除的数组的集合（“删除集合”）。
- i是将要被插入到R中的数组的集合（“插入集合”）[5]。
- d⊆r，即 d 是 r 的一个子集，意味着我们不可能删除任何不存在的数组。
- i 和 r 交集为空（意味着我们不可能插入任何已经存在的数组）。
- d 和 i 的交集也为空。
- d和i是被清晰定义并且唯一的[6]。

以上几点可以很方便地用文氏图来解释清楚，如图 2.2 所示。解释说明：在这个图表中 r、d 和 i 如上面所述，u 则是与它们同类型的全域关系（换句话说，u 是由所有与 R 具有相同关系头的数组所组成的关系，当然也包含那些已经在 R 之内的数组）。请注意 u-r 的差集是 r 的绝对补集（也就是说，u-r 是由所有与 R 具有相同关系头，但不包含在 R 之内的数组所组成的关系）。

当然，删除集合 d 有可能为空，在这种情况下，之前的赋值效果就相当于一个纯粹的 INSERT 操作。或者当插入集合 i 为空时，赋值效果就相当于一个纯粹的

[5] 严格来说，d 和 i 当然不仅仅是“数组的集合”，而是和 r 同一类型的关系。

[6] 请参看附录 A 中对此命题的证明。

DELETE 操作。又或者，d 和 i 都为空，那么赋值操作的效果就会退化为一个“空操作”$R := R$。

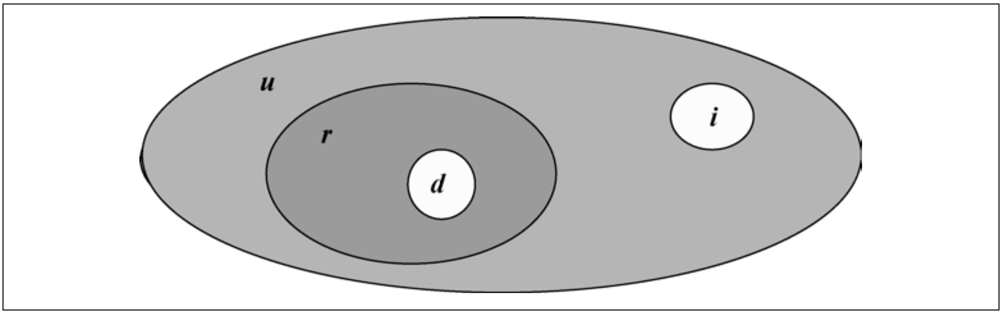



图 2.2　删除集合 d 和插入集合 i

根据以上所说的，原来的赋值$R: = rx$实际上等价于下面的“多重”赋值（“多重赋值”详见本节后面的详细介绍）[7]。

```
DELETE d FROM R , INSERT i INTO R
```

如此一来，虽然我前面曾经说过这类关系赋值其实是我们唯一需要的更新操作，但是我们还是可以始终把它想成 DELETE 和 INSERT 这两个操作（从感觉上来讲，好像还是这样想起来方便一些）。这是因为当我们遇到一个很可能发生的随机的针对关系变量 R 的多重赋值时，包括前面例子中这种用公式表达的比较清晰的对 R 的 UPDATE 操作，我还是建议将赋值转化为一个对 R 的 DELETE 操作和一个对 R 的 INSERT 操作，毕竟删除集合 d 和插入集合 i 是明确定义的、交集为空的，和独一无二的。注意：由于这里已经讲过了比较清晰的 UPDATE 操作，所以接下来我只有在遇到值得说的地方才会再次讨论这个问题。同样，我也会忽略 d 和 i 在实际运算中所碰到的大多数问题。和我其他大多数作品一样，在本书中我首要关注的是先要让大家明白这个理论，而不是首先关注实际执行中出现的问题。当然，我并不是说实际操作当中遇到的问题不重要。Au contraire（法语：恰恰相反），实际上检查实施操作的可行性对确保本理论的正确性是至关重要的。

## 语法要点

我在这里再次重申，关系赋值 $R: = rx$ 在逻辑上等价于：

```
DELETE d FROM R , INSERT i INTO R
```

请一定注意，正是因为 d 和 i 的交集为空，所以无所谓 DELETE 或者 INSERT 哪个

[7] 真正在 Tutorial D 中对于多重赋值的语句应当是 DELETE R d, INSERT R i。

操作“先进行”。（关于这个问题我一会儿还要再多说一点，即在多重赋值的情况下独立操作的完成情况。）所以，我们等于可以这么说：原始赋值逻辑上与下面任意一个关系表达式等价。

```
WITH ( R := R MINUS d ) : INSERT i INTO R
WITH ( R := R UNION i ) : DELETE d FROM R
```

我们甚至可以说它也和下面任意一个关系表达式等价。

```
WITH ( DELETE d FROM R ) : INSERT i INTO R
WITH ( INSERT i INTO R ) : DELETE d FROM R
```

如果你不熟悉上面所使用的 WITH 操作的话，我推荐你看看《SQL and Relational Theory》。

其实还有更棒的。Tutorial D 还额外支持 DELETE 和 INSERT 的变体，分别叫作 I_DELETE（“既有值 DELETE”）和 D_INSERT（“无交集 INSERT”）。使用 I_DELETE，会在你试图删除一个不存在的数组（也就是最开始没有出现的数组）时报错；类似的，使用 D_INSERT 会在你试图插入一个已经存在的数组（也就是已经出现的数组）时报错。因此，我们可以说原赋值 *R*: = *rx* 逻辑上等价于以下任意关系表达式。

```
I_DELETE d FROM R , D_INSERT i INTO R
D_INSERT i INTO R , I_DELETE d FROM R
```

为了简化问题让大家易于理解，在本书之后的部分我还是会以传统的 DELETE 和 INSERT 操作作为例子进行讲解，并且我还会在全文中应用下面这个假设，**请一定注意！**尝试删除不存在的数组不是错误，尝试插入存在的数组也不是错误。

## 多重赋值

“第三宣言”需要的，也是我们已经提过的 Tutorial D 肯定支持的，是一种多重赋值的方式，这种方式允许许多彼此独立的赋值操作“同时进行”。例如：

```
DELETE ( S WHERE SNO = ‘S1’ ) FROM S ,
DELETE ( SP WHERE SNO = ‘S1’ ) FROM SP ;
```

解释：首先，请注意第一行末尾的逗号，它意味着两个 DELETE 都是同一个整体声明的一部分。第二，如我们所知，DELETE 其实就是赋值，所以上面这个“双 DELETE”其实就是下面这个双赋值的缩写。

```
S := ... , SP := ... ;
```

这个声明分别给关系变量 S 和 SP 各赋了一个值，它们都是同一个操作的一部分。我们概括来讲，多重赋值的语义有下面 2 条。

- 首先，在等式右侧的独立赋值的源表达式进行计算。
- 然后，那些独立赋值操作被执行。[8]

我们来观察一下，正是因为所有的源表达式都要先进行计算，然后独立赋值再执行，所以那些独立赋值都不会被其他运算结果影响，也因此它们被执行的顺序也跟运算结果无关（如果愿意，你甚至可以把它们的执行想象成平行发生的，或"同时进行"的）。另外，多重赋值从语义上来讲被当作"原子级"操作，也就是说在这类赋值发生的"过程中"不会进行任何一致性检查（这也正是为什么"宣言"从开始就要求对操作的支持）。注意：一致性约束在本章后面会有详细讨论。

## 语义不是语法

我前面已经说过，任何一个关系赋值都可以等价为一个 DELETE 操作加上一个 INSERT 操作，当删除集合 d 和插入集合 i 是被准确定义，交集为空，并且独一无二时。但是请注意下面这个重点，两个不同的赋值可以使用完全不同的语法，但却对应相同的删除集合和插入集合，下面我为大家详细说明（在这里要感谢 Hugh Darwen 提供了这个例子）。

假设一个关系变量R只有两个属性，K和A。我们令{K}为单键，然后，令K与A的类型都是INTEGER，并令R只包含两个数组，（1，2）和（3，−2）。[9]现在我们来看下面两个显式的UPDATE操作。

```
UPDATE R : { K := K + A } ;
UPDATE R : { A := -A } ;
```

请注意，具体来说第一个操作是"键UPDATE",而 第 2 个则不是，因此，如果一个格式，如ON UPDATE K...的补偿操作已经被定义（在SQL环境下很可能发生），那么我们可以预测这个操作会与第一个UPDATE有关，而与第 2 个无关。但是很容易看出，如果我们给R设定具体的初始值，那么这两个UPDATE的效果完全等价。实际上，它们都等价于下面这个赋值。[10]

---

[8] 如果遇到两个或更多独立赋值拥有相同的目标变量，那么这个定义就需要被改进，不过目前我们还用不到这些。可以参看附件 A 来获得更多解释说明。

[9] 我将在这里和全书其他地方都使用这种简单的表示法来表示数组，以便大家阅读。

[10] 这个赋值在等号右侧的表达式是一个"关系选择器调用"，其实就是一个关系"字面量"。如果你想获得对相关内容的详细解释，请参考《SQL and Relational Theory》。

```
R := RELATION { TUPLE { K 1 , A -2 } , TUPLE { K 3 , A 2 } } ;
```

换句话说，删除集合 *d* 对这两个 UPDATE 而言是完全一样的，而插入集合 *i* 也是如此。（练习：这两个集合它们到底是什么？）因此很明显，我们想要的是对于补偿操作而言，它应当是由相应的删除集合和插入集合适时地调用，而不是由碰巧列出更新表达式的语法武断地选择。

## 2.3 一致性约束

每个关系变量都受制于一系列一致性约束，我们也可以把它们简称为约束。首先，我们在"关系和关系变量"这一节中已经知道，任何一个给定的关系变量都被约束为一个确定的类型（具体来说，是一个确定的关系类型），也就是说，在定义关系变量的时候，这个类型也就确定了。例如，我们再来看看供应商关系变量S的定义。[11]

```
VAR S BASE RELATION
  { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
    KEY { SNO } ;
```

如你所见，这个定义清晰地表明了关系变量 S 属于类型 RELATION {SNO CHAR, SNAME CHAR, STATUS INTEGER, CITY CHAR}。而且，具体来看它还表明了属性 SNO、SNAME、STATUS 和 CITY 分别是 CHAR、CHAR、INTEGER 和 CHAR 这几种数据类型。注意：后一种约束——给每个独立属性的约束，有时候被称为"属性"约束，本段就是一个很好的例子。

我再次重申一遍，任何一个关系变量（任何一个属性更是如此）都被约束为某种类型。但是关系变量同时也会被很多其他的约束所限制，这些约束通常都是使用Tutorial D的表达式明确写出CONSTRAINT声明从而实现的。[12] 下面举几个简单的例子希望能帮助大家理解（如果你需要更多的解释说明，请参阅《SQL and Relational Theory》）。

```
CONSTRAINT CX1 IS_EMPTY ( S WHERE STATUS < 1 ) ;

CONSTRAINT CX2 IS_EMPTY ( S WHERE CITY = 'London' AND STATUS ≠ 20 ) ;

CONSTRAINT CX3 IS_EMPTY ( ( S JOIN SP ) WHERE STATUS < 20 AND PNO = 'P6' ) ;
```

现在"第三宣言"中要求所有的约束都必须在声明边界上被满足（"立即检查"）。

---

[11] 关系变量 S 是个基础变量，这是肯定的。关于视图的情况，请见第 3 章。

[12] 这里也有例外，有些约束可以用 KEY 和 FOREIGN KEY 的方式来表达，但即使是这些约束本质上也仅仅是约束的缩写形式，它们都可以使用 CONSTRAINT 清晰地进行声明，只不过会比较冗长。请参看《SQL and Relational Theory》一书来获得关于这一点更深入的讨论信息。

换句话说，从逻辑上讲，对于任何有潜在可能会违反“宣言”的声明，在它约束时都会被立刻重新检查一次。[13]或者我们也可以说得有趣一点，它们是“在分号处”被检查的。因此，与SQL标准版本或其他特定的SQL产品不同，一致性检查从来都不会被推迟到执行结束或者执行COMMIT（提交）操作的时候。

最后请注意，有时为了方便，我们对于给定的关系变量 R 的“全局”约束，这里要强调是针对“关系变量”的全局约束，指的是所有涉及到关系变量 R 的约束的逻辑 AND（与）。而有时为了方便，我们对于给定的数据库的“全局”约束，这里要强调是针对“数据库”的全局约束，指的是所有涉及到该数据库中任意关系变量的约束的逻辑 AND（与）。

## 更新每次都是集合更新

尽管这个观点尽人皆知，但还是值得我们再次强调，在关系模型中的每次更新都是集合更新（更好的说法是：每次都是“关系”更新）。我们换句通俗点的话来说，INSERT 操作给目标关系变量插入了一系列数组，DELETE 操作将一系列数组从目标关系变量中删除。而更广义地来讲，关系赋值将一系列数组值赋给了目标关系变量。当然，我们经常会这么说（例如）“插入某某数组”，确实，我在这本书里也会时不时这样说，但是这种说法确实是很不严谨（虽然比较方便）的。不论如何，这里我要说的重点在于一致性约束的检查要在“所有的”更新步骤（包括与之相关的补偿性操作，如果有的话）全部完成后进行。换句话说也就是从逻辑上讲，一个集合级别的更新一定**不能**被当作一系列单独数组级别的更新来对待。

## 两个重要的原则

现在我要在此特别强调 2 个重要的原则，这 2 个原则对于巩固更新（尤其包括视图更新）的概念非常重要。第 1 条原则也被称为**黄金法则**。

- **定义：黄金法则**规定所有的数据库都不能违反它自己的全局数据库约束。

显然，由此条法则我们可以立刻推导出下面的规则：所有的关系变量都不能违反它自己的全局关系变量约束。简单来讲，就如同我所强调过的，约束必须在声明边界上被满足。我再次重申，请尤其注意，这条法则无论是对视图，还是对基础关系变量，都同样适用。

第 2 条原则被称为“赋值原则”。

---

[13] 我在这里简单一点说：约束的效果很强大，一些关系选择器调用确实会让所需类型的关系检查得“更迅速”，甚至把检查作为得出关系选择器调用结果的流程中的一环。同样，请参看《SQL and Relational Theory》一书来获得关于这一点更深入的讨论信息。

- **定义**："赋值原则"规定当值 v 被赋给变量 V 之后，比较表达式 v=V 的结果必须为 TURE。

这条原则适用于所有的赋值操作，实际上，相信你也察觉到了，它基本上就是赋值本身的"定义"，不过就本文而言，它尤其能够适用于关系赋值。当然也请再一次注意，这条法则无论是对视图，还是对基础关系变量，都同样适用。

## 2.4 关系变量谓词

在第 1 章中，当我介绍抛砖引玉的例子（供应商与零部件数据库）时，曾提到了这样一件事：

表 S 的内容列出了签过合同的供应商。每个供应商有一个唯一的供应商编号 SNO、可能不唯一的供应商名字 SNAME（尽管图 1.1 中的样本值恰好是唯一的），以及状态值 STATUS 和位置 CITY。

当然，在第 1 章中的"表 S"似乎应该叫作"关系变量 S"更加准确，不过这并不是重点。重点在于：第一，目前的文字代表了关系变量 S 的意义，并且更容易被用户理解；第二，这种意义的措辞实际上是"谓词"（有时候也说"解释"或者"预期解释"，但在本书中我还是主要使用"谓词"这种说法）。换句话来讲，可以说每一个关系变量都有一个相对应的谓词，我们称之为这个关系变量的"关系变量谓词"，而关系变量谓词的本质是用户对与之对应的关系变量的理解。下面我们来举个例子，这里我选择声明关系变量 S 的谓词。

S：供应商 SNO 是已经签约的，名字为 SNAME，有状态 STATUS，位置在 CITY 城市。

下面是关系变量 P 和 SP 的谓词。

P：零部件 PNO 在公司内使用，名字为 PNAME，有 COLOR 颜色和 WEIGHT 重量，并且存储在 CITY 城市。

SP：供应商 SNO 供应零部件 PNO，数量为 QTY。

作者注：[14] 也许我应该对我在本书中所使用的"谓词"这个词语多做一些说明。首

[14] 这段注文是从《Database Design and Relational Theory》中截取过来的，我进行了少量编辑。注意：我曾经在前言就提醒过你，在本书中我会使用《Database Design and Relational Theory》作为缩写的形式来表示引用我的《Database Design and Relational Theory: Normal Forms and All That Jazz》（奥莱利，2012）。

先一点，在SQL中这个词被广泛地用来表示布尔值或真值表达式（比如比较谓词、IN谓词、EXISTS谓词等），所以你可能对它已经很熟悉了。但是，虽然这种用法在SQL上并不是完全错误的，但它确实滥用了一个非常通用的词汇，这个词汇在数据库环境中非常重要，并且给了它一个特别具体的意思，这也是为什么我决定不沿用这种用法的原因。

第二，为了信息准确我应该来解释一下，谓词并不是像上面我们写的，就是自然语言中的几个句子。相反，谓词是由这些句子所决定的。换个说法就是关系变量 S 的谓词该是什么就是什么，不管你是用英语或者西班牙语，还是其他任何语言来表达，它的核心内容都是一样的。但是在后面的文章中，为了表达简明，我会假设谓词就是 1 句话，一个（通常）用自然语言表达的句子。（类似的标注也会用到命题上，看下面。）

最后，我已经解释了这个词的意思，尽管有前面一段的解释，但是你还是需要知道，对于谓词“到底”是什么，即使逻辑学家们也很难有非常一致的答案。尤其是有些作者认为谓词是一个单纯的概念，它本身并无意义，而认为我上面所写的应当是预期解释，而不是谓词。我不想在这里对这些事情做过多的争辩。如果想看更多的讨论，我推荐你去看“What’s a Predicate?（什么是谓词）”这篇文章，你可以在 Hugh Darwen 和我合著的著作《Database Explorations: Essays on The Third Manifesto and Related Topics》（Trafford，2010）中找到它。

你可以把谓词想成是一个“真值函数”。和所有的函数一样，它也有一系列参数。当它被调用的时候会返回一个结果，而（因为是真值函数，所以）结果是 TURE 或者 FALSE。那么在关系变量 S 的谓词这个例子中，这些参数就是 SNO、SNAME、STATUS 和 CITY（对应的显然是这个变量的属性），它们的值分别是相应的类型（分别是 CHAR、CHAR、INTEGER 和 CHAR）。当我们引用调用该函数时，逻辑学家们会说当我们“实例化谓词”时，我们把一组确定的值代入参数。假设我们代入的值分别是 S1、Smith、20 和 London。那么可得：

供应商 S1 签约了，名字是 Smith，状态是 20，位于城市 London。

这句话其实是一个“命题”，在逻辑上它并没有参数，也无法得出一个明确的“ture”或“false”的结果。（当然对于手中这个例子，命题为真，或者至少我们这样假设。）这把我们引入另一个非常重要的原则中，它就是闭域假设（The Closed World Assumption）。

- **定义：**令关系变量R拥有谓词P。那么闭域假设（The Closed World Assumption，CWA）规定（a）当数组t在T时刻出现在R中，那么我们就假设P中对应于t的实例p在T时刻为真；相反地，（b）当数组t与R拥有相同的关系头，但在T时刻并

未出现在R中，那么我们就假设P中对应于t的实例p在T时刻为假。换句话说，当且仅当数组t在给定的时刻满足R的谓词时，数组t才会在该时间出现在R中。[15]

举个例子，我们再次参看图 2.1，数组（S1，P1，300）出现在关系变量 SP 中，因此供应商 S1 可以提供 P1 零部件，数量 300 就成为一个“真正的事实”。但是如果我们从该变量中删除该数组，那么我们就会说（实际上）供应商 S1 可以提供 P1 零部件，数量 300“不再是个事实”。

到目前为止，我们讨论了谓词和关系变量的具体关系。不过，实际上目前所有的概念都自然而然地向关系型表达式延伸。例如，考虑到供应商在除了CITY以外的所有属性上都有投影（注意Tutorial D对投影的语法）。[16]

```
S { SNO , SNAME , STATUS }
```

这个表达式表明一个关系包含所有形如（s，n，t）的数组，而一个形如（s，n，t，c）的数组现在出现在关系变量 S 中，CITY 值是 c。换句话说，就是结果包含且仅包含所有对应下面谓词中真值实例的数组。

**这里存在着这样的城市 CITY**，供应商 SNO 已签约，名字是 SNAME，状态为 STATUS，并且位于城市 CITY。

这个谓词代表了一个表达式——一个关系表达式 S{SNO,SNAME,STATUS}。观察这个表达式你会发现，这里有 3 个参数，而对应的关系本身有 3 个属性。CITY 并不是该谓词的参数，而是被逻辑学家们称为“约束变量”，因为实际上它被“这里存在着这样的城市 CITY”这句话所“量化”了。注意：这里可能有一个更清楚的方式来说明这点（也就是，谓词有 3 个参数，而不是 4 个），那就是这个谓词效果上等价于下面这一个。

供应商 SNO 已签约，名字是 SNAME，有状态 STATUS，并且位于某处（意思是它位于某个城市，但是我们不知道具体是哪一座城市）。

注意类似的方式可以应用到任意可能的关系表达式中。具体来说就是：任意关系表达式rx总是有一个对应的意义，或谓词；而且rx的谓词总是可以被表达式中的关系

[15] 我想强调“当且仅当”。这里至少有一个重要的推论。具体来说就是令关系变量 R1 和 R2 分别拥有谓词 P1 和 P2，那么当 P1 和 P2 同时被同一个数组 t 满足时，t 必须同时出现在 R1 和 R2 中。（虽然这是一种经验法则，但它可能是一个很好的方式来确保 P1 和 P2 足够特殊以排除前面的可能性。但是请注意，在第 4 章以及第 9～11 章我们将看到一些蓄意违反这个原则的情况出现。）

[16] 这里我要说明，我们发现一件很便利的事情，就是在 Tutorial D 的所有操作中，可以把最高优先级设给投影。

变量的谓词决定，与表达式中关系操作的语义保持一致。[17]

作者注：再次重申，除了 CITY 之外，所有供应商的属性的投影都可以用 Tutorial D 表示为形式 S {SNO,SNAME,STATUS}。而为了更加方便地表达，Tutorial D 也允许在表达式中利用被移除的属性代替被保留的属性进行表达。因此，以上面的表达式为例，它和表达式 S {ALL BUT CITY}是等价的。在适当的情况下，在 Tutorial D 范围内类似的情况可以应用于所有的操作。

就我目前为止对关系变量谓词的描述来看，好像它们很不正规，实际上它们只是在自然语言中对关系变量拥有特定的属性（当然，是特定的类型）这一事实的陈述和表达。不过我们也可以使用更加正式一点的表达方式。例如，给定供应商关系变量 S 有属性 SNO、SNAME、STATUS 和 CITY，并且给定{SNO}是该关系变量的单键，那么我们说相应的谓词应该如下面这样。

```
is_entity  ( SNO ) AND
has_SNAME  ( SNO , SNAME ) AND
has_STATUS ( SNO , STATUS ) AND
has_CITY   ( SNO , CITY )
```

假设数组 t 出现在关系变量 S。那么这个谓词表达的意思是：

- SNO 在 t 中的值确定了一个实体。
- 这个实体拥有一个 SNAME 的属性，而该属性的值是由 SNAME 在 t 中对应的值给定的。
- 这个实体同时也拥有一个 STATUS 的属性，它的值是由 STATUS 在 t 中对应的值给定的。
- 这个实体同时也拥有一个 CITY 的属性，它的值是由 CITY 在 t 中对应的值给定的。

当然，系统并不知道这个实体，即被 SNO 确定的实体，其实就是现实世界中的一个供应商，实际上是一个“已签约”的供应商，也不知道它的意思是拥有 SNAME 属性、STATUS 属性或 CITY 属性。

[17] 因此，我们可以将闭域假设加以扩展。在一个给定的时间内，我们设一个命题 p 在对应数组 t 并且在数组 t 满足下列条件时为真：（a）数组 t 在该时间（前面所说的给定的时间）内出现在某个数据库的关系变量中，或者（b）数组 t 在该时间内出现在某个关系中，并且可以由该关系推导出数组 t 就是这个数据库关系变量的值。我们换一句简单点的话来说，就是所有由数据库声明或说明的均为真，其他均为假。

类似地，我们也可以说除了 CITY 之外所有供应商属性的投影的谓词为：

```
is_entity  ( SNO ) AND
has_SNAME  ( SNO , SNAME ) AND
has_STATUS ( SNO , STATUS ) AND
EXISTS CITY ( has_CITY ( SNO , CITY ) )
```

最后，在结束本节的时候我要向大家坦白，到目前为止我所讲的所有关于关系变量谓词的内容都故意地进行了简化。不过当然，我相信这样略有简化的内容对于做介绍来说是足够的了。而对于更详细的内容——“真实的故事”，或者至少是比现在的“真实一点”的，我建议你参看我的另一本书《Logic and Databases: The Roots of Relational Theory》（Trafford，2007）第 4 章（“闭域假设”）。

## 2.5　MATCHING，NOT MATCHING 以及 EXTEND

就本书的主体部分而言，我当然会讨论一些关系代数的运算符（及其代表的操作）的细节，而基本上我会认为你们理所当然已经对这些运算符及其操作（如果你需要，可以在附录 B 中找到它们的定义）很熟悉了。但是这里有一些我经常提及的特定操作你们可能并不熟悉；它们被 Tutorial D 所支持并且很有用处，但是在大多数常见的教材中却并未被讨论过（可能因为它们并不能直接被 SQL 支持）。第一个我们要说的就是 MATCHING，下面是它的定义。

- **定义**：表达式 rl MATCHING r2 所返回的关系等于表达式 rl JOIN r2 所返回的关系，都投影到 rl 的属性上。

这里有一个例子（“求得最近供应过至少一个零部件的供应商”）。

```
S MATCHING SP
```

给定我们经常使用的样本值，那么结果应该像下面这样：

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |

Tutorial D 同时也支持一个“非”格式的操作 NOT MATCHING。下面是它的定义。

- **定义**：表达式r1 NOT MATCHING r2 所返回的关系等于表达式r1 MINUS (r1

MATCHING r2)所返回的关系。[18]

这里有一个例子（“求得最近没有供应任何零部件的供应商”）。

```
S NOT MATCHING SP
```

给定我们经常使用的样本值，那么结果应该像下面这样：

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S5 | Adams | 30 | Athens |

另外一个我想给大家介绍的操作是 EXTEND。EXTEND 有 2 种形式。第 1 种形式的定义如下。

- **定义：**令关系 r 没有任何一个属性叫作 A。那么表达式 EXTEND r :{A := exp} 所返回的关系的关系头是扩展了属性 A 之后的 r 的关系头，关系体则是数组 t，t 是扩展了 A 的值之后的 r 的数组，而 A 的值是通过将 r 的数组应用表达式 exp 计算后得出的值。

那么现在我们来看，假设关系 P 中零部件的重量一般使用磅为单位，现在我们想用克为单位来显示这些重量信息。克与磅的换算关系是 454 克为 1 磅，所以我们可以这么写：

```
E XTEND P : { GMWT := WEIGHT * 454 }
```

给定我们经常使用的样本值，那么结果应该像下面这样：

| PNO | PNAME | COLOR | WEIGHT | CITY | GMWT |
|---|---|---|---|---|---|
| P1 | Nut | Red | 12.0 | London | 5448.0 |
| P2 | Bolt | Green | 17.0 | Paris | 7718.0 |
| P3 | Screw | Blue | 17.0 | Oslo | 7718.0 |
| P4 | Screw | Red | 14.0 | London | 6356.0 |
| P5 | Cam | Blue | 12.0 | Paris | 5448.0 |
| P6 | Cog | Red | 19.0 | London | 8626.0 |

EXTEND 的第 2 种形式主要瞄准的是“假设情况”查询。换句话说，它可以直接浏览特定变化带来的效果，而不需要对目标对象做出实际变化（可能然后再撤消）来实现。这里有个例子（“要是零部件的重量单位使用的是‘克’而不是‘磅’会

[18] 我做这个标记是为了说明著名的关系差操作——Tutorial D 中的 MINUS 是 NOT MATCHING 的一个特例。具体来说就是如果关系 r1 和 r2 是相同类型，那么 r1 NOT MATCHING r2 会简化为 r1 MINUS r2，这一点可以很容易确认。因此，相对来说 NOT MATCHIG 比 MINUS 更加基础。

怎么样？”)。

```
EXTEND P : { WEIGHT := WEIGHT * 454 }
```

这里的重点在于在大括号中的赋值目标属性并不像上一个例子中那样是一个“新”的属性，而是一个已经存在于特定关系（在这里这个关系就是关系变量 P 目前的值）中的属性。整体来讲，这个表达式被定义成与下面所列出的表达式等价。

```
( ( EXTEND P : { GMWT := WEIGHT * 454 } ) { ALL BUT WEIGHT } )
                                        RENAME { GMWT AS WEIGHT }
```

正如你所见，这个展开式中使用了 Tutorial D RENAME 操作。我想这个操作已经差不多人尽皆知了，当然如果你需要的话还是可以在附录 B 中找到相应的定义。更多关于它的讨论请参阅《SQL and Relational Theory》。

让我们用第 2 种形式的定义来作为本节的结尾。

- **定义：**令关系 r 拥有属性 A。那么表达式 EXTEND r :{A := exp} 所返回的关系的关系头与 r 相同，关系体则是数组 t，t 是由替换 A 值后的 r 的数组得出的，而替换后的 A 值则是该数组经由表达式 exp 计算得出的。

# 2.6 数据库与数据库变量

这是本章的最后一节。在本节中，我希望你们能注意一个重要的事实，正如关系值与关系变量（relvars）之间有逻辑区别一样，数据库值和数据库变量（dbvars）之间也存在着逻辑差异。下面一段引自《宣言》那本书。

第 1 版的“宣言”将数据库的本质（也就是数据库的值）与数据库变量进行了区分……[同时也]指出未明确定义的词汇“数据库”应该就是明确地代表数据库的值，并介绍了作为“数据库变量”的缩写来使用的词汇“dbvar”。虽然我们依然相信这个区别本身是真实有效的，但随即我们发现其实它与“宣言”所要讲的其他方面内容几乎没什么关系。因此我们决定，为了文章的易读性，还是在文中使用传统的说法。[19]

于是，当我们“更新某些关系变量”（在某个数据库内部）时，实际上我们更新的

[19] 换句话说，我们继续使用“数据库”这个词，时而表示数据库变量，时而表示数据库值，并且我们压根不使用 database variable 或者 dbvar 这两个词。但是后来我们发现，因为种种原因我们没有设计这部分是一个非常错误的决定。（做一些你知道的逻辑上错误的事情通常不是个好主意。）至于我在本书中将计划如何来使用这个术语，请参看本节剩余的部分。

是相应的数据库变量。（为了表述清晰，在本节的其余部分我都将使用数据库变量这个词。）例如，Tutorial D 声明

```
DELETE ( SP WHERE QTY < 150 ) FROM SP ;
```

“更新设备供应关系变量 SP”其实是更新了整个供应商-零部件数据库变量——对这个数据库变量的“新的”数据库值和“老的”值基本一样，除了特定的“设备供应”数组被移除了。换句话说，有时候我们会说一个数据库“包含变量”，即相应的关系变量，其实这种说法很笼统，而且很不正式。更加正式的并且更准确的描述方式应该是这样的：

**数据库变量就是数组变量。**

数组变量对该数据库变量中的每一个关系变量都拥有一个唯一的属性（并且没有其他属性），并且每个属性都有关系值。我们举例来说，在供应商-零部件数据库中，我们可以将整个数据库变量想象成一个数组变量（也可以用“tuplevar”表示），数组类型如下：

```
TUPLE { S RELATION { SNO CHAR , SNAME CHAR ,
                     STATUS INTEGER, CITY CHAR } ,
        P RELATION { PNO CHAR , PNAME CHAR ,
                     COLOR CHAR , WEIGHT RATIONAL , CITY CHAR } ,
        SP RELATION { SNO CHAR , PNO CHAR , QTY INTEGER } }
```

让我们暂时称供应商-零部件数据库变量（或者也可以叫它数组变量）为SPDB。[20] 那么上面我们刚刚看到过的DELETE声明可以作为下面的“数据库”（以及数组）赋值。

```
SPDB := TUPLE { S ( S FROM SPDB ) ,
                P ( P FROM SPDB ) ,
                SP ( ( SP FROM SPDB ) WHERE NOT ( QTY < 150 ) ) } ;
```

解释说明：这个赋值右侧的表达式表示一个数组拥有 3 个属性分别叫作S、P以及SP，每一个属性都拥有关系值。[21] 在数组内，属性S的值就是当前关系变量S的值，属性P的值就是当前关系变量P的值，而属性SP的值则是在当前关系变量SP的值的基础上，减去那些数量小于 150 的数组。

我们来看另一个例子，这里我们再次使用“双删除（DELETE）”，我在本章早些时候介绍多重赋值概念的时候曾经使用过。

[20] 如果我们想要像这个例子中一样写出清晰的数据库赋值，那么数据库变量就必须拥有用户可见的名字，但在 Tutorial D 中它们没有（至少按照目前的定义来说）。

[21] 实际上，上面这个表达式是一个数组选择器调用。再次说明，如果你想获得关于这个的更多说明，请参看《SQL and Relational Theory》。

```
DELETE ( S WHERE SNO = 'S1' ) FROM S ,
DELETE ( SP WHERE SNO = 'S1' ) FROM SP ;
```

现在我们观察这个关系赋值，很明显是个多重赋值，会发现它逻辑上等价于下面的“单一”数据库（以及数组）赋值。

```
SPDB := TUPLE { S  ( ( S FROM SPDB ) WHERE NOT ( SNO = 'S1' ) ,
                P  ( P FROM SPDB ) ,
                SP ( ( SP FROM SPDB ) WHERE NOT ( SNO = 'S1' ) ) } ;
```

换句话说，目标变量全部都是同一个数据库（或者说同一个数据库变量）中的关系变量，多重赋值其实仅仅就是一个语法工具，它可以让我们将逻辑上的“数据库”赋值表达成一个一系列独立“关系”赋值的组合（请参看下面关于此点更多的讨论）。而一个“单一”的关系赋值，例如由只针对一个单独的数据库变量进行的一个单独的赋值所组成的“多重”赋值——仅仅是个特例。**从根本上来说，事实上目前对数据库更新来讲，数据库变量是唯一真正的变量**。这就是说，数据库更新其实“总是”对某些数据库变量进行更新，而对单一一个数据库关系变量的更新仅仅是特例而已。

根据上面所有的内容，如果目标关系变量是一个数据库关系变量（就是说，如果这个目标关系变量是某个数据库中的关系变量），那么即使一个“单一”关系赋值其实也是一个数据库赋值。当然，这个数据库赋值像所有的赋值一样，都必须遵循“赋值原则”，并且正由于它是一个数据库赋值，它也同样需要遵循“**黄金法则**”。

我们前面讲到的这些想法太重要了，它值得我们再次复习一遍，只是说法有轻微的不同。首先，数据库变量是数组变量，数据库（也就是在给定的时间，给定的数据库变量所具有的值）是所有属性都有关系值的数组。第二，给定一个关系赋值如下面格式。

```
R := rx
```

（这里R是一个关系变量参考量，从语法上讲就是一个关系变量的名称，指出了一些数据库关系变量，而rx是一个关系表达式，表示关系r与R具有相同类型）这个关系变量参考量R其实是一个“伪变量”参考量（请立刻参看本段下面的注解）。换句话说，关系赋值其实是针对相对应数据库变量（再次重申它就是数组变量）的一个部分进行“替换”的简称化表达方式。因此我们推断出“关系变量”（至少是在数据库中的关系变量）并不是真正的变量，它们只是一些让人觉得很便利的虚构的变量，能够制造出这种假象：数据库或数据库变量可以一点一点地、一个关系变量一个关系变量地被更新。我们进一步可以推断出，在某种意义上，“关系赋值”（不

管是不是多重的）也同样是个方便的假象，具体来说就是它是一个虚拟操作，让我们感觉好像可以通过一系列针对这个数据库内的单独的关系变量的更新，来实现对数据库变量的更新。[22]

关于伪变量的注解：从本质上讲，伪变量是这样一个概念，或者是这个概念的象征性的名字，它看起来像是个可操作的表达式，但不同的是它并不表示任何值；相反地，它出现在某些赋值中的目标位置，来表示“变量的一部分”。我们举个例子，令 X 为一个 CHAR 类型的变量，并令 X 目前的值是“antimony”。那么赋值表达式 SUBSTR(X,5,3) := ‘nom’的效果就是“替换”X 的第 5～7 个字符，将“mon”替换成“nom”，并因此将 X 的“旧的”值（“antimony”）完全替换成“新的”值（“antinomy”）。这个赋值左侧的 SUBSTR(X,5,3)就是一个伪变量参考量，而这个赋值本身则是下面表达式的简化版：X := SUBSTRX(X,1,4) || ‘nom’ || SUBSTR(X,8,1)，在这 里符 号“||”表示字符串连接。注解完毕。

为了本书的易读性和定义的明确性，虽然有违我的最佳判断，但我还是决定在本书余下的部分不再使用数据库变量和“dbvars”这两个专用术语，至少不会用很多次，而是使用大家熟悉的“数据库”，希望我要表达的意思在上下文中总是很清楚。读者要擦亮眼睛。

[22] 从这个讨论中我们可以看出多重关系赋值是极其重要的（在缺少明确的对数据库赋值的支持的情况下），实际上它对于后面章节要讨论的一些特定的视图更新规则来说非常的关键。因此在这种情况下，我要说，请原谅我在这里指出，很不幸，至少 SQL 不支持这个操作。

# 第3章 视图概念：近距离观察

距离让景色更迷人。

——Thomas Campbell:《Pleasures of Hope》(1799)

任何一门学科都有自己的未解之谜。我们首先以数学为例，这里有黎曼假设，该假设在 150 多年间没有人能完全证明或否定；在计算机科学中，我们有“P=NP？”问题，40 多年过去了，它依旧未被完全解答；在物理学和宇宙学中，我们有关于“万物理论”的漫长研究，然而它至今依旧——在大多数人的概念中，可能不是所有人的——只是停留在研究过程中；在数据库理论领域，我们有视图更新的问题。在这里我并不是要说视图更新问题和黎曼假设、“P=NP？”问题或者具有猜想性质的“万物理论”是同一级别的问题，[1] 我要说的是它具有相当高的实践重要性和理论价值。因此在本章中，我将给大家介绍一些必要的基础理论，以便在后面我们向着这个问题发起系统性的攻击。

下面我将从一个老生常谈的观察开始，视图就是关系变量——一个虚拟的关系变量，让我们更确切一点来说，一个关系变量“看起来和感觉上”就像一个真实的或者基础的关系变量，但是（与真实的或者基础关系变量不同的是）它不能离开其他关系变量单独存在。也就是说，它就是由其他关系变量来定义的，或者说推导出来的。下面是其定义。

**定义：**视图是被推导出来的、虚拟的关系变量。视图V在T时刻的值是经由特定的表达式（视图定义表达式）在该时刻计算得出的。当V被定义后，该表达式应该是

[1] 我的一位审阅者坚持说它们就是同一级别的问题。也许它是，但我不想就这点争论下去。

确定的，并至少包含一个关系变量。[2]

请看下面的例子（它就是第 1 章中“伦敦供应商”的视图 LS 的 Tutorial D 定义）。

```
VAR LS VIRTUAL ( S WHERE CITY = ‘London’ ) ;
```

众所周知，视图可以被看作一个类似于通往定义它的那些关系变量内部的“窗口”，这就意味着针对这个视图所进行的操作会“真的”对它底层的关系变量进行操作。换句话说，无论是更新，还是检索，数据库管理系统都应当可以支持用户对视图进行操作，通过衡量那些操作并转化成合适的针对基础关系变量的操作来实现，而这些基础关系变量由视图的最终定义来决定。（我之所以在这里说“最终定义”是因为如果视图真的可以表现得像基础关系变量一样，那么有一件事情我们确定可以实现，那就是可以在视图基础之上再定义视图，无论几层都可以。）因此，我们看个例子，如下的表达式：

```
LS WHERE STATUS > 10
```

（这个表达式可以理解为一个对视图 LS 的检索操作）对应的是：

```
S WHERE CITY = ‘London’ AND STATUS > 10
```

而下面在 LS 上的更新操作（实际上是个删除操作）：

```
DELETE ( LS WHERE STATUS > 10 ) FROM LS ;
```

对应的是：

```
DELETE ( S WHERE CITY = ‘London’ AND STATUS > 10 ) FROM S ;
```

通过上面所述，我们可以清楚地看出视图和关系变量之间存在着逻辑差异。不过根据可交换性原则，这个差异并不会影响用户。我再重申一次我们的目标，那就是从用户来角度来说，让视图“看起来和感觉上”就像基础关系变量一样，对于其更新也是一样。（基础关系变量和视图之间的这种差异与数据库管理员和数据库管理系统是有关系的，但是我需要再次重申，它与用户并没有关系。至少这就是我们想做到的。）换句话说，最理想的情况就是对于给定视图的用户来说，他们根本不会知道他们对于该视图所进行的操作会对应到底层关系变量的操作上。实际上理想情况是用户甚至应该都不知道他所操作的是一个视图，也不知道这个视图后面还有底层关系变量存在。

---

[2] 它必须包含至少一个关系变量，否则这个视图就不会是一个关系变量，而仅仅是一个关系常量。（关于这点更多的讨论请参看第 13 章 ）

顺便说一句，关于“真实”对比“虚拟”关系变量的说法可能对大家有些误导，其实视图并不比基础关系变量更不（或更加）“真实”，不过现在我先不做更多解释，一部分原因也是它确实反映了 Tutorial D 定义视图时的语法使用情况，正如我们已经看到的。但是，在本章的后面，我还是要针对这个不是很好的用词选择和语法多做一些说明。

### 视图更新问题

下面我将正式说明“视图更新问题”到底是什么：现在假设对任意的视图给定任意一个更新操作，或者应该说给定任意一个更新请求，那么其“问题”就在于根据视图的定义来确定到底需要对关系变量执行什么操作。（请注意这个问题的回归特性。我会在第 14 章中针对这一点做更多说明。）当然，这里我有两个默认的假设，那就是对于这些操作所作用的底层关系变量而言（a）它们都是真实存在的，并且（b）它们都是被成功定义的。在未来的章节里我也会继续对这两点进行讨论。

注意：当然我也许应该说明，还需要假设系统能够支持视图。从个人角度讲，我确实认为视图应该被支持，原因我会在 3.2“独立数据”这一节中做出解释。更深入来说，既然我说（a）视图就是关系变量，（b）关系变量就是变量，且（c）变量可以被更新，那么（d）视图必然可以被更新。但是我想你所站的角度也可以是说这里一开始就根本没有视图要被支持的明确需求！如果是这样，那么我建议你立刻停止阅读本书，既然你已经到了这个地步，那么我想我应该增加这个假设。

## 3.1　视图是伪变量

现在我想把重点放到这个话题上来，即视图就是关系变量。换言之，如前所述，视图就是变量[3]，也就是说根据变量的定义，它们是可以被更新的。（作为变量是可以被更新的，同样的，如果它可以被更新，那么它一定是个变量。）也就是说，总体上讲，将视图作为关系赋值的目标对象必须是可行的。我们就用前面章节定义好的视图LS来举例说明，对它来说，下面的DELETE操作应该是合法的。

```
DELETE ( LS WHERE SNO = ‘S1’ ) FROM LS ;
```

当然，这个 DELETE 操作是下列关系赋值的简化操作。

```
LS := LS WHERE NOT ( SNO = ‘S1’ ) ;
```

而这个赋值操作又是下面这个展开版本的简化版（只不过下面的展开版在目前

[3] 至少是和它背后的基础关系变两一样！请参看我在第 2 章最后一节中对这个观点的标注。

Tutorial D 的定义下是非法语句）。

```
( S WHERE CITY = ‘London’ ) :=
               ( S WHERE CITY = ‘London’ ) WHERE NOT ( SNO = ‘S1’ ) ;
```

由此我们可以得出，从更新的角度来看，视图就如伪变量一样[4]（请回忆一下上一章我们所讲的，伪变量基本上就是这样一种概念，它们看起来像是可执行的表达式，但是却出现在赋值操作的目标对象中，这正是我们在这里所遇到的情况）。而伪变量就是变量（不太严格地讲），因此请大家一定要注意，在本书余下的部分中，当提到“变量”这个词的时候，它其实是应当包括伪变量在内的。

## 3.2 数据独立性

请看下面这段文字，引自 Codd 在 1970 年的著名论文的摘要（“A Relational Model of Data for Large Shared Data Banks，”《Communications of the ACM 13》，第 6 号，1970 年 6 月）。

未来的大型数据银行用户[原文如此]一定要免于遭受必须了解机器内数据结构（内部表示法）的苦恼……用户在终端上的操作和……应用程序都应该不会受到内部表示法，甚至是一些外部表示法变化所带来的影响。

文中第 1 句话是说我们需要数据在物理上的独立性，第 2 句话是说我们同时也需要数据在逻辑上的独立性。事实上，数据独立性正是Codd最初开始介绍和推广关系模型的最主要的原因之一。[5] 数据物理独立性指的是我们可以在对用户的访问体验毫无影响的情况下改变数据的物理存储和访问方式。类似的，数据逻辑独立性则是指我们可以在对用户的访问体验毫无影响的情况下改变数据逻辑上的存储和访问方式。而这 2 种独立性的实际意义都是为了“保护客户投资”，在这里我们所谓的投资指的是对现有的已经存在的各种培训、各类应用和数据库（扩展点说甚至包括目前已有的数据库管理系统）的投资。

那么现在我们来具体看看视图。广义地说，数据逻辑独立完全可以实现，视图为其实现提供了最好的手段。请容我细细道来。正如我在本书前言中所提到的，我们可以不失概括性地假设，客户所看到的数据库全部都是由视图组成的。所以，在数据的逻辑独立性实现的前提下，如果底层基础关系变量发生了变化，那么客户所见的视图对底层基础关系变量的映射就会自动做出相应的调整。下面我们来看一个例子

[4] 当然，基础关系变量也是一样的。同样的，请参看我在第 2 章最后一节中对这个观点的标注。
[5] 在这里我要指出当今主流的数据库管理系统仍然不能很好地实现这一点。

（为了简洁我们只做大体介绍）。假设用户最初看到一个视图 V 与供应商基础关系变量 S 完全一致。

```
VAR V VIRTUAL ( S ) ;
```

现在假设由于某些原因——具体原因在这里尚无需深究，我们决定将基础关系变量 S 替换为基础关系变量 LS（“伦敦供应商”）和 NLS（“非伦敦供应商”），于是：

```
VAR LS BASE RELATION /* London suppliers */
  { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
    KEY { SNO } ;

VAR NLS BASE RELATION /* non London suppliers */
  { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
    KEY { SNO } ;
```

那么现在我们要做的是重新定义视图 V，具体如下：

```
VAR V VIRTUAL ( LS UNION NLS ) ;
```

假设系统可以正常地支持操作（尤其是更新操作）在视图上实施（很不幸，就目前的产品情况来说实属一个大胆的假设），那么视图 V 的用户就会完全不受这次针对底层基础数据变动的影响。

现在，我们看到了为什么具备视图更新的能力是如此重要。如此一来就确实有点难以理解为什么这里会有如此多的质疑围绕着（更不用说完全反对）视图应该能够更新这个观点了。事实上，问题（a）恰当地更新基础关系变量从而能够支持视图的更新，理论上来讲等同于问题（b）恰当地更新存储的数据从而能够支持基础关系变量的更新。它们只是在整体系统架构上表现在不同的方面，仅此而已。由此可见，我们必须解决这个问题，否则我们就只能放弃数据独立这个目标。（这里我们说的是逻辑上的数据独立，但是通过上面这个简短的讨论已经足够说明逻辑的与物理的数据独立从本质上讲都是同样的问题。它们的区别也只是在整体系统架构上体现在不同方面而已。）

### 视图为多种目标服务

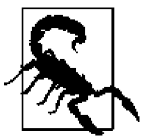
在《SQL and Relational Theory》这本书中，我曾说过视图为 2 种不同的目标服务。具体而言，我是这么说的。

- 我们说，能够定义给定视图 V 的用户，显然知道这个视图是由表达式 exp 来定义的（也就是相应的视图定义表达式）。因此，这个用户可以在表达式 exp 的地方使用名字 V。但是，这仅仅是把它当作缩写来用，并且该用户也清楚地知

道它只是个缩写。而且，该用户可能基本上不会有针对 V 的更新需求，当然，即使发生了这类需求，它们也必须像预期的那样发生。

- 反过来说，假设一个用户仅仅知道 V 的存在并知道它可供使用，但并不知道视图定义表达式，那么对该用户来说，V 应该看起来和感觉上就像是一个基础关系变量，正如我所说（请参看前面关于逻辑数据独立性的讨论）。

就这两个目标来说，第二个更为重要，同时它也是本书所关注的重点。但是请注意，这个目标还会被再细分为两种可能的情况：被既定用户看到的这个视图可以（a）整体上有效地等价于底层基础数据，具体请参看本章后面关于信息等价的讨论，或者（b）等价于底层基础数据的某些子集。后一种情况构成了“信息隐藏”的一种形式。正如第 1 章所标注的那样（请参看第 1 章“规则至上”这一节），被隐藏信息的用户只能看到全局的一部分，而且基本可以肯定，通常都会有一些特定的操作是不允许这类用户运行的。在本章后面部分我们将会看到一些这类情况的例子。



“安全”是使用视图时需要隐藏信息的一个明确原因。例如，某条规则限制某些用户不能看到供应商所在的城市，为了实现这个效果，我们可以通过让这些用户使用一个不包含关系变量 S 的 CITY 属性映射的视图来实现。但是，我认为，这里还有比使用视图更好的方法来控制权限。更好的方法那就是控制哪些用户被允许对哪些数据做什么样的操作（请参看我的著作《An Introduction to Database Systems》第 8 版，Addison-Wesley 出版，2004 年）。换句话说，我认为因为这个使用视图只是为了转移注意力，而且我也不准备在本书中就这个问题做过多讨论。

## 一个心理上的误解

现在让我们来特别关注一下 SQL。事实上，按照 SQL 的说法，只要与用户有关的视图都应该看起来和感觉上跟基表一样，这个说法特别强调了 CREATE TABLE 和 CREATE VIEW 之间的语法差别一直是个错误，至少是心理上的误解。首先误解的第一点，CREATE TABLE 中的关键词 TABLE 指的就是基表，从而（非常不幸地）更加强调了 SQL 界普遍认为的视图与表存在着某种差异这一观点。让我们简明而不失详细地说明这一点：众所周知，SQL 的文档——教科书、产品说明书以及 SQL 标准本身，都无一不使用“表和视图”这种说法（可以说这种说法无处不在）。但很明显的是所有这么说的人都认为表和视图本就是不同的东西，具体来说，他们很可能认为“表”总是指的基表，而且可能还认为表都是实际存储在介质上的，而视

图不是。但最重要的一点是视图“就是”表（或者，我更倾向于说“关系变量”）。因此，任何能够在普通关系变量上执行的操作，都应该可以对视图执行，因为视图本身“就是”正式的关系变量。因此，在本书中，我将会使用术语“关系变量”来表示所有的关系变量，如果要具体针对基础关系变量或者视图来做说明，我会相应地使用更具体明确的词汇来表达。

误解的第二部分：SQL 中 CREATE VIEW 的语法是这样的，在视图和定义视图的表，换句话说，就是相应的视图定义表达式中的表，之间的映射被非常清晰地表述在视图的定义中。于是能看到 CREATE VIEW 的用户就都能清晰地看到这个映射关系，尽管实际上这些用户逻辑上并不需要看到它（这个映射）。理想状态下，其实这些用户根本不需要知道它的存在。由此导致的一个结果就是用户通常会知道这个视图确实就是一个视图，更广义地说，就是用户通常会知道哪些表是基表，哪些表是视图。很讽刺的是由此导致一个更加复杂的问题出现，那就是用户本来需要知道的视图的“架构”并没有被清晰地定义在 CREATE VIEW 中，反而是被从映射关系中推断出来的！（这里所说的“架构”，我指的是视图的每列的名字以及它们相应的数据类型，换句话说，也就是“关系头”，但是“关系头”并不是一个 SQL 名词。实际上 SQL 对于这个概念根本就没有一个专有名词。）

为了避免有人说我对 SQL 的批评不公平，让我澄清一下，目前为止的所有批评，在做出相应的微调后，对 Tutorial D 也同样适用（至少对它目前的版本是这样）。同时也让我说明一下我打算如何解决这些问题。

- 第一，对于用户可知的关系变量的定义（当然）应当具体写明相应的关系头，但不应当用任何方式区分基础关系变量和虚拟关系变量。
- 第二，(a）视图和其定义中关系变量之间的映射关系应当分别写清，并对用户隐藏，就如（b）基础关系变量和物理存储的数据之间的映射关系也是分别写清，但对客户隐藏这种方式一样。

## 3.3 如何预防

在我进一步详细阐述视图应当如何运作之前，我想很有必要在这里先来浏览一下这一领域中业界“目前的工艺水平”（虽然价值不大）。实际上，这个“目前的工艺水平”留下了很多需要完善的地方，实话实说，是非常之差。当然，在SQL标准和目前的商业版本中来说是这样，甚至（很可惜）在某些技术著作中也是如此。那么针对SQL来说，我们经常能够在一个给定的SQL数据库管理系统中找到如下几条规

则[6]。

- 禁止对逻辑上应该可以更新的视图进行更新。
- 允许对“逻辑上不能更新”的视图进行更新。[7]
- 用一种逻辑上错误的方式来执行视图的更新。
- 包含以上全部。

（我要冒险再多说一句，最后这一点是在实际应用中最常遇到的情况。）

让我就这点再多说一些，更具体地说，让我就 SQL 标准再多说几句。我前面已经说过，针对这一部分的标准是有缺陷的。实际上，它不仅仅是有缺陷的，而且是让人非常难以理解的！针对这个部分的标准甚至比其他部分的标准更加让人费解。下面的摘要（是逐字逐句地引自 SQL：2003，2003 版的 SQL 标准）可以给大家展示它有多么复杂。

当且仅当对于每个包含于 QE1 的<查询表达式>或<查询具体描述>QE2 都满足下列条件时，<查询表达式>QE1 是“可更新的”。

a） QE1 包含 QE2 但不包含一个使用了 UNION DISTINCT, EXCEPT ALL，或 EXCEPT DISTINCT 的<查询表达式主体>。

b） 如果 QE1 包含<查询表达式主体>QEB，QEB 使用了 UNION ALL，那么：

  i） QEB 立即包含一个<查询表达式>LO 和一个<查询条件>RO，这样 LO 的一般基本表就不可能同时也是 RO 的一般基本表。

  ii） 对于 QEB 的每一列，分别由 LO 和 RO 定义的表中的优先列全部都可更新，或者全部都不可更新。

c） QE1 包含 QE2 但其中不包含使用 INTERSECT 的<查询条件>。

---

6 当今 SQL 数据库管理系统对视图的更新方式起源于《Views, Authorization, and Locking in a Relational Data Base System》这篇论文，它是由 Donald D. Chamberlin，James N. Gray，和 Irving L. Traiger 在 1975 年于加利福尼亚州阿纳海姆举行的国家计算机大会上发表的（会议记录刊于 AFIPS Press，1975 年 5 月）。在事后看来，我大胆地说，也许有点不公平，在实际应用中对于视图概念常见的一些困惑和混淆都表现在这篇论文的标题上了。

7 在这儿你可能会有点皱眉，因为按照我在第 1 章所说的，“所有的”视图都应该是可以更新的。但是我当时所说的是——而且现在依然是指所有的“关系型”视图都是可更新的。我既不知道，也不关心是否非关系型视图是可更新的——这里我是指那些涉及到非关系型概念的部分，如重复的行或者未知的值（虽然我很肯定无法从逻辑上证明这类视图可以更新）。

d） QE2 是可更新的。

下面是我自己给这个摘要的注释。

- 首先，观察这段摘要你会发现，它仅仅体现了很多规则中的 1 条，而这些规则必须综合使用才能确定某些给定的视图在 SQL 中是否是可更新的。
- 这些规则并不是在一个地方全部给出的，而是被分散到标准的许多不同地方。
- 所有这些规则都需要多种额外的概念和结构的支持，比如，可更新的列、一般基本表、<查询条件>，这些都在标准的其他部分才给出定义。

除此之外，在这段摘要中被定义的这条规则根本就是让人觉得不知所云。我们具体来看，开篇第一句话说的是“对于每个……包含于 QE1 的 QE2”来说，这 4 个条件 a)、b)、c）和 d）必须都满足，可是 b)这一条中根本就没有提到 QE2（？）。

我的批判还没有结束。事实上，由于 SQL 确定了 4 个不同的情况而使得局面更加复杂，具体来说就是 SQL 中的视图是“可更新的”、“潜在可更新的”、“简单可更新的”或者“可插入内容的”。现在，SQL 标准定义了这几个术语，但是并没有给出任何关于这些术语的直观的解释说明，或者这几个术语的命名原因。不过，至少我可以确定“可更新的”指的是 UPDATE 和 DELETE 操作，而“可插入内容的”指的是 INSERT 操作，并且还有一点就是如果视图是不可更新的，那么它是不能被插入任何内容的。但是请注意这种情况，某些视图可能只允许一部分更新操作执行，而不允许其他的操作执行（比如允许 DELETE，但不允许 INSERT），这 也就意味着有可能会出现 DELETE 和 INSERT 无法互为逆操作的情况。以上这些情况（如果它们是事实的话）我至少都会把它们当作潜在的违反“可交换性原则”的情况来考虑。

作者注：这里出现的非对称现象让人觉得很奇怪。一场争论可能由此而发——不是我发起的！那就是你可以针对合并视图执行 DELETE 操作，而不能执行 INSERT 操作，这是因为删除规则很明确（从两个操作数都删除），但是插入规则却不是（我们是向所有的操作数都插入，还是只向其中一个插入？如果只向其中一个插入的话，那是向哪一个进行？）。而你马上就会发现有另一个非常相似的争论说你可以针对交集视图执行 INSERT 操作，但不能执行 DELETE 操作。换句话说，如果某些视图是“可删除，但不可插入的”，那么其他的视图肯定就是“可插入，但不可删除的”（？）。

出于显而易见的原因，我甚至不想去尝试为“SQL 认为可以更新的视图”做精确的特性描述。不过，不严格地讲，它们至少包含以下几种情况。

1．视图被定义为一个单独的基表的约束和/或投影（键保存）。

2．视图被定义为一个一对一或者一对多的双基表连接（在一对多的情况下，只有多的那一方是可更新的）。

3．视图被定义为对于两个不同的基表的 UNION ALL 操作或者 INTERSECT 操作。

4．1～3 的特定组合。

对于第 1 种情况，我还能说得更精确一些。那就是在下列所有条件都满足的情况下，SQL 视图一定是可更新的。

定义表达式要么是（a）一个简单的 SELECT 表达式（也就是说，不是 UNION、INTERSECT 或者 EXCEPT 表达式），要么是（b）一个“明确的表”，也就是说，按照 TABLE <表名字>格式来写的表达式，这逻辑上等价于一个简单的 SELECT 表达式。注意：为了简单明了起见，在后文中我假设情况（b）会自动转化为情况（a）。

- 在 SELECT 表达式中 SELECT 语句具体是 ALL，而不是 DISTINCT。
- 对“*号”扩展之后，SELECT 后面的用逗号分隔的每一项就是简单的列引用。
- 在 SELECT 表达式中 FROM 语句使用 FROM T [AS ...]这种格式，其中 T 是一个可更新的表的名字（既可以是基表，又可以是可更新的视图）。
- 在 SELECT 表达式中如果有 WHERE 语句，则它不包含 FROM 语句引用表 T 的子查询。
- SELECT 表达式不包含任何 GROUP BY 或者 HAVING 语句。

但即使是这些限制条件，SQL 都不能正确处理，这要感谢它对下面 4 点缺乏理解：（a）**黄金法则**，（b）赋值原则，（c）约束推断（请参看紧接着的 3.4 节“约束和谓词”），以及（d）SQL 允许出现未知值（null）以及重复行这个事实。难道我们不能做得更好些吗？

## 3.4 约束和谓词

视图就是关系变量并因此受到关系类型的约束。说得更详细一点就是任意给定的视图V的类型，实际与V的定义表达式类型一致。[8]我们再次使用LS（“伦敦供应商”）

[8] 请参看附录 B 中对任意关系表达式类型的相关解释。

来举例说明，下面是针对LS的Tutorial D定义。[9]

```
VAR LS VIRTUAL ( S WHERE CITY = ‘London’ ) ;
```

这个视图是一个“约束视图”，它的定义表达式调用了关系变量 S 上的一个特定约束操作，因此它的类型就与关系变量 S 相同，也就是 RELATION {SNO CHAR, SNAME CHAR, STATUS INTEGER, CITY CHAR}。

而更广泛地说，视图就是关系变量，因此要受到我在第 2 章中称为全局关系变量约束（针对本句话中的关系变量）的制约。但是，对于视图 V 的全局关系变量约束是（至少部分是）“推导”出来的约束；它是（或至少部分是）从 V 的定义中所涉及到的关系变量的约束中推导出来的，与定义表达式中涉及到的关系操作的语义保持一致。对于视图 LS 这个例子来讲，全局关系变量约束就是下面几条的逻辑 AND。

a．对关系变量 S 的全局约束，以及

b．在视图定义表达式中所规定的约束条件(CITY = ‘London’)。[10]

换句话说，那就是视图 LS 受制于所有对关系变量 S 起效的约束。例如，{SNO}是键这个约束，**并且还有**城市的值是伦敦这个约束。请注意，也正因为如此，“黄金法则”对视图与对关系变量同样适用。

现在让我们姑且使用“视图约束”这个词来表示所有对视图适用的约束。那么，即使视图约束可以像前面这个例子一样被推导出来，这也并不能说它们不需要清楚地声明（另一方面讲，系统并没有“智能”到可以自动确定这些约束到底是什么的程度）。因此，为视图来做出清晰的约束声明显然是个合理的要求。尤其是下面两条，更是非常合理的诉求：（a）应当可以在视图的定义中包含清晰的 KEY 和 FOREIGN KEY 的详细描述说明，以及（b）应当可以允许 FOREIGN KEY 描述中的目标关系变量是一个视图。下面的例子展示了（a）中所说的 2 种可能性。

```
VAR LS VIRTUAL ( S WHERE CITY = ‘London’ )
     KEY { SNO }
     FOREIGN KEY { SNO } REFERENCES S ;
```

视图就是关系变量，因此也拥有关系变量谓词，下面我们就来说一说谓词中的参数，

---

[9] 我会继续使用常规的 Tutorial D 语法对视图进行定义，尽管我对 3.2 节的语法表达感到担心。

[10] 一般来说，我们应当清楚，在任意给定时间，由约束表达式 *rx* WHERE *bx* 表示的关系必须满足所有对由关系表达式 *rx* 表示的关系**以及**由布尔表达式 *bx* 所表示的约束所适用的约束（换句话说，在这里我们可以把 WHERE 理解为 AND）。注意：由 *bx* 表示的约束在这里是一个限定条件。请参看附件 B 中关于此概念的详细定义。

这些参数与相应的关系变量，也就是视图的属性是一对一的关系。但是适用于视图 V 的谓词也是“推导”出来的，它是从定义 V 所涉及的关系变量的谓词推导而来的，与定义表达式中涉及到的关系操作的语义保持一致。其实我们已经知道这件事了，因为我们从第 2 章已经得知每一个关系表达式都有一个对应的谓词，那么对于视图来说它的定义表达式当然也会有对应的谓词。让我们再次用视图 LS（“伦敦供应商”）作为例子。这个视图是关系变量 S 的限定条件，因此它的谓词是 S 的谓词和下面具体的限定条件的逻辑 AND。

供应商 SNO 已经签约，名称是 SNAME，有状态 STATUS，并且位于城市 CITY

并且

城市 CITY 是伦敦。

或者说得更通俗点。

供应商 SNO 已经签约，名称是 SNAME，有状态 STATUS，并且位于 CITY 伦敦。

不过请注意，后面一种说法会淡化 CITY 是参数这件事情。确实它是个参数，但是相应的争论却总是发生在常量“伦敦”上。（也正是因为这个原因，我才在第 1 章中标注，视图 LS 更现实一点的版本应该避开 CITY 属性。在这里我选择不回避是为了保证这个例子简单易懂。不过，我在第 5 章会对这件事有更多讨论。）

作者注：如果你觉得第 2 章中所写的关于谓词更正式一点的形式比较好的话，那么视图 LS 的谓词可能写成下面这样会让你更容易接受。

```
is_entity  ( SNO ) AND
has_SNAME  ( SNO , SNAME ) AND
has_STATUS ( SNO , STATUS ) AND
has_CITY   ( SNO , CITY ) AND
CITY = ‘London’
```

## 多元约束

有另一个和约束有关的事情我需要在这里做简短的讨论，不过它与谓词刚好相反（虽然这件事情不仅视图约束会涉及到，而且是对所有的约束都要做的）。首先，让我们回忆一下第 2 章里面出现过的这个简单的例子。

```
CONSTRAINT CX1 IS_EMPTY ( S WHERE STATUS < 1 ) ;
```

当然，这个例子是用Tutorial D写的表达式，不过在这里为了能够快速说明白，我

想把同一个约束（或者至少是这个约束中的布尔表达式部分，就是从约束的名字 CX1 后面一直到结束的分号中间这一段）通过关系运算表达出来。[11]

```
NOT EXISTS x ∈S ( x.STATUS < 1 )
```

或者另一种等价的方式：

```
FORALL x ∈S ( x.STATUS ≥ 1 )
```

在这些公式中，子表达式“*x*∈S”（这个表达式可以读作“有这样的 x，是 S 当前值中的一个数组”）将变量 x 定义为一个“区间变量”，“区间落在”关系变量 S 的当前值上。换句话说，任意给定时间内被允许的 x 的值都是关系变量 S 的值中出现的数组。

作为未来示例的基础，让我们同时来看看视图 LS（“伦敦供应商”）和视图 NLS（“非伦敦供应商”）。下面是一些用关系运算表达的适用于这两个视图的约束。

```
FORALL x ∈ LS ( x.CITY = ‘London’ )
FORALL x ∈ NLS ( x.CITY ≠ ‘London’ )
```

现在我们在本节中看到的都是些“单变量”约束，因为它们每一个都只含有一个区间变量。那么下面这个例子则是一个“多元”约束（它基本就是第 2 章中的约束 CX3）。

```
FORALL x ∈ S ( FORALL y ∈ SP
     ( IF x.STATUS < 20 AND x.SNO = y.SNO THEN y.PNO ≠ ‘P6’ ) )
```

作者注：顺便说一句，请注意在这里我们说的多元约束与《SQL and Relational Theory》中所说的“多关系变量”约束有着逻辑上的区别。多元约束指的是一个约束中引入了两个或以上的区间变量，而多关系变量约束则是指一个约束中引入了两个或以上关系变量。现在我们可以肯定地说多关系变量约束同时也是多元约束，但是反过来就不对了，在多元约束中的区间变量有可能只涉及一个关系变量（例如下一个示例）。

下面是另外一些多元约束的例子（它们都基于关系变量 S、LS、NLS）。

- 在 LS 和 NLS 中没有任何供应商号码。

```
FORALL x ∈ LS ( FORALL y ∈ NLS ( x.SNO ≠ y.SNO ) )
```

- 对于 S、LS 和 NLS 来说，{SNO}都是键（为了简单起见，我这里只列出了对

[11] 如果你需要更多关于本节中所说的关系运算表达式的信息，可以参看《SQL and Relational Theory》。

关系变量 S 的约束)。

```
FORALL x ∈ S ( UNIQUE y ∈ S ( x.SNO = y .SNO ) )
```

> 如果你不熟悉 UNIQUE 也没关系，它可以读作“存在着这样的一个……”。

- LS 中的{SNO}是外键，引用 S 中的键{SNO}。

```
FORALL x ∈ LS ( UNIQUE y ∈ S ( x.SNO = y.SNO ) )
```

- S 等价于 LS 和 NLS（交集为空）的合并。

```
FORALL x ∈ S ( UNIQUE y ∈ LS ( x = y ) OR UNIQUE y ∈ NLS ( x = y ) ) AND
FORALL y ∈ LS ( UNIQUE x ∈ S ( x = y ) AND
FORALL y ∈ NLS ( UNIQUE x ∈ S ( x = y )
```

- 在任意给定的时间，LS 等价于当 CITY 值为伦敦时对 S 的约束。

```
FORALL x ∈ LS ( UNIQUE y ∈ S ( x = y ) AND
FORALL y ∈ S ( IF y.CITY = ‘London’ THEN UNIQUE x ∈ LS ( x = y ) )
```

现在（最后）我们终于来到了这段可能有点长的讨论的重点。在第 1 章中，我介绍了有关补偿性操作的一些观点，这些补偿性操作是用户需求之外的，为了防止某些违反一致性约束的情况发生而由数据库管理系统自动执行的额外的更新。那么，如果这里说的约束是多元约束的话，就真地需要某些补偿性操作发生了，我们在后面的几章中也会看到。

## 3.5 信息等价

在第 1 章中第一次提到信息等价时，我举了个这样的例子（让我整理一下语言），一个数据库设计单独引入关系变量 S，在信息上等价于，将关系变量 LS 和 NLS（分别为“伦敦供应商”和“飞伦敦供应商”）共同引入整合后的效果。同时我也强调了单独引入关系变量 LS 的设计是无法信息等价于上面提到的任何一种情况的（当然，单独引入关系变量 NLS 也不行）。下面我将更详细地解释我的这些论点。

当我说到一个数据库设计单独引入关系变量S在信息上等价于将关系变量LS和NLS共同引入时，我所表达的观点，我相信大家清楚我要表达的意思，是说任何可以被其中一种方式所表述的命题同时也可以被等价的另一种方式所表述。我们换一种直观的方式来说，就是这两种设计当且仅当它们可以代表相同的概念时，才可以说在

信息上等价。[12] 这也正是下面的定义的本质。

- **定义：**[13] 令DB1 和DB2 是若干关系变量的集合。[14] 那么DB1 和DB2 当且仅当适用于DB1 和DB2 的约束可以实现任何一个能够被DB1 所表述的命题都可以被DB2 所表述时，信息等价。反之亦然。

让我们再次将关系变量 S、LS 和 NLS 作为例子考虑。令 DB1 与关系变量 S 所含的关系变量一致，DB2 与关系变量 LS 和 NLS 整合后所含的关系变量一致，这样一来，很明显我们可以使用 DB1 中的单个关系变量来定义 DB2 中的多个关系变量，于是有：

```
LS ≝  S WHERE CITY = 'London'

NLS ≝ S WHERE CITY ≠ 'London'
```

（“≝”这个符号的意思是“定义为”）。同时很明显我们也可以使用 DB2 中的多个关系变量来定义 DB1 中的单个关系变量，于是有：

```
S ≝ LS UNION NLS
```

除此之外，我想还有一点不用细讲也能够明显看出的事情是对于每个适用于 DB1 的约束，这里总有一个类似的约束可以适用于 DB2，反之亦然。关于这点我只给一个简单的例子：对于 DB1，在关系变量 S 中没有任何两个不同的数组可以拥有相同的供应商号码；对于 DB2，有两点，（a）关系变量 LS 和 NLS 的合并交集为空，（b）在这个合并中没有任何两个不同的数组拥有相同的供应商号码。所以我希望你能认同，至少直观地来看，每个能被 DB1 所表述的命题都能够被 DB2 所表述这一点还是很合理的，反之亦然；换句话说，DB1 与 DB2 之间就是信息等价。

前面我们的讨论提出了一些观点，现在我们看看接下来的另外几点。令 DB1 和 DB2 分别是信息等价的关系变量的集合，并且令它们目前的值分别为 db1 和 db2。现在假设，在给定时间内能够被 DB1 所表述的命题应当与在相同时间内能够被 DB2 所表述的命题完全相同。那么：

---

[12] 从这里我们可以推导出另一个可以被用于形容这个概念的词汇是“表达”等价。

[13] 请注意这个定义在一定程度上被简化了。在第 14 章中我还有更多关于这个问题的解释。不过在那之前，我们以这里所给出的定义为准。

[14] 我在这里使用代号 DB1 和 DB2 是想说我们可以更直观地将每组关系变量连同它们的约束想象成一个数据库，或者在这里更准确地说是一个数据库变量。事实上在后续的章节中，当我在谈论信息等价的例子时，为了简洁我有时可能会将所讨论的关系变量集合当作数据库或者数据库变量来讲。注意：这里我要特别说明，对于代号 DB2，我并没有特指可能与它重名的某些商业产品，这点请一定了解。

- 这些变量的值 db1 和 db2 应当相应地互为信息等价（当然能被 db1 所表述的命题也能被 db2 所表述，反之亦然）。换句话说，信息等价这个概念非常实用，既可以用于几组关系之间，也可以用于几组关系变量之间。

- 如果 db1 和 db2 互为信息等价，那么这里一定分别存在映射 M12 能够将 db1 转化为 db2，和映射 M21 能够将 db2 转化为 db1，这两个映射应该由，或可以由关系代数的操作公式所表示（当然它们应该是互逆的）。反过来讲，如果上述映射存在，那么 db1 和 db2 就互为信息等价。而如果类似映射对于每组 DB1 的当前值 db1 和 DB2 的当前值 db2 都分别存在的话,那么 DB1 和 DB2 本质上也是信息等价的。（当然，这几点加起来，要比原始定义所说的更复杂一些。）

- 对于在 db1 上的任意一个查询 Q1,都一定存在一个在 db2 上的查询 Q2 产生与 Q1 相同的结果（显然反之亦然）。注意：从这个结论我们可以导出对于 IS_EMPTY (Q1)和 IS_EMPTY (Q2)来说，要么全真，要么全假，因此正如之前所说的，对于任意一个 DB1 上的约束 C1，在 DB2 上一定存在一个 C2 与之对应，反之亦然。

- 令 U1 为 BD1 的更新，可以产生一个 DB1 的“新”值 db1’，那么在 DB2 上一定存在一个更新 U2 可以产生一个 DB2 的“新”值 db2’，且 db1’与 db2’信息等价（反之亦然）。图 3.1 展示了这一点。

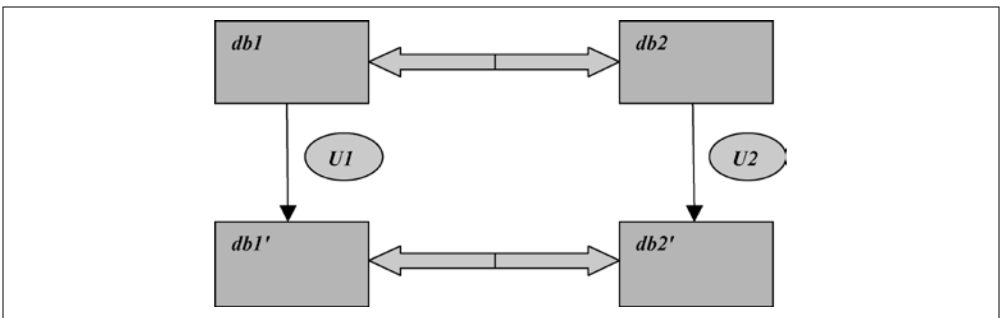



图 3.1　信息等价与更新

相比前面所讲的而言，假设 DB1 和 DB2 不是信息等价的。那么就会有在 DB1 上的操作——查询和更新在 DB2 上找不到对应等价操作的情况出现，反之亦然。（注意，有的时候，如果 DB1 和 DB2 确实不是信息等价的，那么它们的当前值 db1 和 db2 既可能互为信息等价，也可能不为信息等价，但是总体上来说，它们应该不为信息等价。）

现在我们来看上面所有的结论在具体例子中的应用，在这个例子中 DB2 只由 DB1

中关系变量的视图组成。换句话说，如果 DB2 中“只有视图”，并且这些视图信息等价于 DB1,那么 在 DB2 上的任意一个更新在 DB1 上都会有一个对应的等价更新，反之亦然。在接下来的几章中，我会更细化地考虑这种情况。也就是说，我会向大家详细展示如何用对 DB1 中关系变量的更新来实现对 DB2 中视图的更新。

那么，如果DB2 只由DB1 中关系变量的视图组成，但是DB2 与DB1 并“不是”信息等价的话会怎么样呢？[15] 在这种情况下，首先请注意，正是由于DB2 中的关系变量都是由DB1 中的关系变量推导而来的，故没有任何能够被DB2 所表达的命题不能被DB1 所表达。或者，我们换个方式来想，任何能够被DB2 所表达的命题一定都能被DB1 表达，但反过来却不对。因此总地来说，有些在DB2 上的更新操作可以通过在DB1 上的更新实现。与此同时，也一定有一些在DB2 上的更新操作不能用这个方式实现（因为可用信息不足），那么这些更新一定是失败的。那么下面几章我们所要讨论的内容结构也因此产生。

1．当信息等价确实成立时，DB2（视图）应当总可以通过一种逻辑上正确的方式被更新，而我会尽可能小心地解释操作规则。

2．即使当信息等价不成立的时候，我们有时也能够应用视图更新规则（可能只能部分应用），那么我也会详细地讨论这种情况。**但我必须强调，在这种情况下实施的更新可能导致如下结果，就是更新操作所应具有的可预测性和清晰的定义，有时可能无法具备，甚至可能是不可接受的。**正因如此，我会给大家（可能是某个用户、数据库管理员，甚至可能是数据库管理系统）留出在这种情况下对是否允许更新操作进行选择的机会。

当视图更新规则完全不起作用的时候，当然我会解释什么时候它们会失效，那么正如我说过的更新操作必定会失败。注意：关于这个情况我确实还有一些更深入的解释，但是我将把这些内容放到第 5 章的最后去讲。

## 3.6 小结

在最后我想用一些与本章内容关系不大的几个问题来结束本章。首先，我想说说关于专有名词的问题。这本书当然是关于视图的，因此我希望当我使用“视图”这个

[15] 在《Database Design and Relational Theory》这本书中，我主张这样的 DB2 在逻辑上是不成立的（或者至少是不完整的），实际上在这种情况下它并不具备表达我们所要表达命题的能力。David McGoveran 会说它没有具备“表达完整性”。此外请注意，我们这里所说的情况正是早前在第 1 章所提到的有关“信息隐藏”的例子：DB2 的用户只看到了全局的一部分，因此可以想见这里一定有一些操作不允许他们执行。

词的时候不需要再强调这一点，我这么做是有原因的，因为原本这个词是被定义好的。但是很不幸，最近几年无论是在学术界，还是在商界都出现了一些专有名词上的混淆。正如我们知道的，视图在传统认知中是一个被推导出来的关系变量，它们显然是由定义它们的关系变量（基础关系变量和/或其他视图）推导而来的。然而，还有另一种被推导出来的关系变量，叫作“快照”。[16] 正如其名字所讲，快照，虽然是被推导出来的，但其实它是一个真实的变量，而不是虚拟的，这就意味着它不仅仅被其他关系变量定义，而且，至少概念上来讲，也被它自己的那部分数据副本所定义。例如（临时发明的语法）：

```
VAR LSS SNAPSHOT ( S WHERE CITY = ‘London’ )
    KEY { SNO }
    REFRESH EVERY DAY ;
```

定义一个快照就好像执行一个查询，除了以下几点不同。

- 查询的结果会被用具体的名字（这个例子中就是 LSS）作为一个只读关系变量存储在数据库中。注意：这里所说的“只读”，指的是在这个关系变量上不允许任何更新操作，但这并不包括周期性的“刷新”，请看紧接下来的一点。
- 这个快照会被周期性地（在例子中是 EVERY DAY<每天>）“刷新”，也就是说它的当前值会被弃用，然后查询被再次执行，而新的查询结果会成为快照的新值。当然，其他的 REFRESH 选择也有可能——例如，EVERY MONDAY（每周一）、EVERY 5 MINUTES（每 5 分钟）、EVERY MONTH（每个月）等。这里特别要说的是 REFRESH ON EVERY UPDATE 意思是快照与推导出它的关系变量实时同步。

因此，在这个例子中，快照 LSS 代表了至多 24 小时之前的数据。

快照在数据仓库中、分布式系统里，以及许多其他环境中非常重要。在所有这些地方，其基本原理都是应用程序可以经常允许，有些地方甚至是要求，得到某个具体时点的数据。报告类应用和会计类应用都是很典型的例子。这类应用通常要求数据要冻结在某个适合的时刻（一般来说是一个记账周期的结尾），而快照可以在不锁定其他应用的前提下实现这种冻结。

[16] 目前就我所知，“快照”这个词第一次出现是在由 Michel E. Adiba and Bruce G. Lindsay 所撰写的“Database Snapshots”（IBM 研究报告 RJ2772，1980 年 3 月 7 日）这篇论文中。该词也同样出现在其扩展版，由 Michel Adiba 所撰写的“Derived Relations: A Unified Mechanism for Views, Snapshots, and Distributed Data”（Proc. 1981 International Conference on Very Large Data Bases,法国，戛纳，1981 年 9 月）中。

到目前来说好像还不错。可问题是快照必须不被当成快照（至少在某些情况下），而是被当作“实体化的视图”。但它们并不是视图！视图完全不应当被实体化。特别是针对视图的操作应当通过映射，从而在定义它的关系变量上执行合适的操作来实现（正如在介绍本章内容时所解释的那样）。[17]因此，“实体化的视图”根本就是一个自相矛盾的词组。更可怕的是现在在某些领域中，“视图”经常不合时宜地被用来表示“实体化的视图”。以至于在原来的认知中，我们很可能不再能用一个恰当的词来表示视图本来的意思。如我所说，在本书中我一定会用“视图”表示它原本的意思，但是一定要小心在其他地方这个词很有可能是别的意思。

那么第二点如下。可能某些读者知道，我已经思考视图更新问题有一段时间了，确实，我试图让视图更新“变正确”有些年头了，而这本代表着我最新想法的书距离我在这个领域出版的第一本著作已经比较久远了。因此，如果本书观点与我早前的著作有什么分歧的话，那么我的态度应当以本书所表达的为准。这里为大家列出我早前的作品以便备用。

- “File Definition and Logical Data Independence”（与 Paul Hopewell 合著），Proc.1971 ACM SIGFIDET Workshop on Data Definition, Access, and Control，加利福尼亚州，圣迭戈（1971 年 11 月）
- “Updating Views”，《Relational Database: Selected Writings》（Addison-Wesley 出版，1986 年）
- “Updating Union, Intersection, and Difference Views”和“Updating Joins and Other Views”（都是与 David McGoveran 合著的），《Database Programming & Design 7》，第 6 号（1994 年 6 月）和第 8 号（1994 年 8 月）；在《Relational Database Writings 1991-1994》中再版（Addison-Wesley 出版，1995 年）
- “View Updates”，《An Introduction to Database Systems》（第 8 版，Addison-Wesley 出版，2004 年）中的第 10.4 节
- “View Updating”，收录于 Hugh Darwen 与我合著的《Databases, Types, and the Relational Model: The Third Manifesto》第 3 版，Addison-Wesley 出版，2006 年），附录 E
- “The Logic of View Updating”，《Logic and Databases: The Roots of Relational

[17] 确实某些特定的产品会使用实体化代替操作映射作为实施手段，具体来说，是作为在特定“复杂”视图上实施特定“复杂”检索的手段，但这个事实与目前的讨论没什么关系（其实它完全是个烟雾弹）。

Theory》（Trafford 出版，2007 年）

- “How to Update Views”，收录于 Hugh Darwen 与我合著的《Database Explorations: Essays on The Third Manifesto and Related Topics》（Trafford 出版，2010 年）

在列表最开头的两条比较不尽如人意，很是令我汗颜，不过其他的（我相信）至少已经步入正轨，虽然可能或多或少有些地方还是不够清晰（按年代来算，越早的文章越会存在更多的不清晰之处，这从某种意义上来看可能也代表了我对这个问题的研究进程）。

我同时要列出 David McGoveran 在这个领域里的专利。

- “Accessing and Updating Views and Relations in a Relational Database”，美国专利号 7263512 （2007 年 8 月 28 日）

- “Computer-Implemented Method for Translating among Multiple Representations and Storage Structures”，美国专利号 7620664（2009 年 11 月 17 日）

本书中所持的观点与这几项专利中的观点有些相似（并且受专利观点的影响很大），而我必须澄清一下它们在细节上还是有很大不同的。

根据前文所写的内容也许很容易能看出，在这个领域中我确实多次改变过观点，我承认这点，这没什么大不了。不过在这里我也要为自己辩护，引用 Bertrand Russell 的话：

有人控诉我已把改变观念当作习惯……我自己并不为[这个习惯]而感到丝毫羞愧。有哪个从 1900 年以来的唯物论者敢发梦说他的想法在过去半个世纪从未改变过？我所看重并且竭尽全力去追寻的哲学是科学的，因此在追寻的过程中总会有些新知识出现而让之前的错误无所遁形，任何一位明眼人都能明白这一点。关于这点我曾经说过，无论什么时候，我都不会像神学家们诉说他们教派的神迹那样来阐述我的事实。最多，我只会将当时最合乎情理的想法告诉大家……如果后面的研究显示它不需要修改，那我反而会感到奇怪。[这种想法不会]被当作教皇的公示一样拥有法力，而仅仅是我竭尽全力经过清晰而认真地思考后所得出的果实。我明确表态，以上所说的已经成为了我的目标。

这些美妙的注脚，无需我做任何解释，都是摘自 Russell 自己的书《The Bertrand Russell Dictionary of Mind, Matter and Morals》（编辑，Lester E. Denonn，Citadel Press 出版，1993 年）的前言。

让我再重复一遍，本书所代表的是我的最新想法。但我并没有说我在这里所说的是我关于这个课题的遗言。这里仍然有一些比较模糊的结论——确实有，这里有一个或两个观点很有理由会被挑战（当我们讲到它们的时候，我会明确地公布这几个观点，以便大家能够清晰了解）。但是我很乐观，我相信这些模糊的问题会在一个合适的时间被明确地解决，而且我也不认为这些问题之中会隐藏有什么能破坏我的理论根基的东西。说实话，我在这个时间以这种形式出版本书的原因之一也是想获得一些富有建设性的建议。当然，如果出现合理建议，我很乐意参与到讨论中去。不过我可能还是要强调修饰语“建设性的”……可能我在这里显得有些保守，这是因为过去在这个领域我已经见过太多的负面批判，而我更希望有更多积极的对话发生。

让我们用最后一点来结束本章：很不幸，视图更新这个话题确实很容易让人迷惑，即使在它不存在争议的时候也一样。一些难点会横在我们的研究之路当中，无法避免地成为研究的障碍，并且很容易让人在争议和讨论中渐渐失去方向。尤其是我们很容易忘记，特别是在范例当中，哪些关系变量是基础关系变量，哪些是视图。保持头脑清晰很重要！让我再说一次：读者要擦亮眼睛。

第 4 章

# 限制视图

更新限制让矛盾去无踪。

——Anon：《Where Bugs Go》

本书余下部分的内容主要由一个调查组成，这个调查是探究有什么因素是在各种关系操作（让我们用回常用的说法）中从始至终一直“贯穿”其中的。因此本章也可以被视为是一个系列的开头章节。在本章中，我会带领大家仔细研究更新限制视图的问题（也就是，更新“贯穿”于限制操作之中）。不过请注意，研究这个主题时会出现很多具有重要意义的问题，因此从某种意义上来说，本章也可以算作是第 3 章的一个延续。我将从对第 1 章中抛砖引玉例子的近距离观察开始讲起（很抱歉，这里会有一点重复内容，但这是近距离探究所必须的部分）。

## 4.1 “抛砖引玉”再回顾

我们一起来复习一下，第 1 章中我们用来抛砖引玉的例子从供应商-零部件数据库中引入了基础关系变量 S(“供应商”)，以及针对该关系变量的两个限制视图 LS(“伦敦供应商”）和 NLS（“非伦敦供应商”)。下面是用 Tutorial D 给出的定义。

```
VAR S BASE RELATION
  { SNO CHAR , SNAME CHAR , STATUS INTEGER , CITY CHAR }
    KEY { SNO } ;

VAR LS VIRTUAL ( S WHERE CITY = ‘London’ )
    KEY { SNO } ;

VAR NLS VIRTUAL ( S WHERE CITY ≠ ‘London’ )
    KEY { SNO } ;
```

正如我们在第 3 章“信息等价”中所了解到的，至少从概念上我们可以清晰地排布所有的因素，以使得关系变量 S 是某数据库 DB1 中的唯一关系变量，关系变量 LS 和 NLS 则是另一个数据库 DB2 中仅有的两个变量，那么 DB1 和 DB2 为信息等价。但在这里我们先不着急讨论这一点。

现在我们来看看它们的定义，{SNO}是这 3 个关系变量的一个键（其实还是个单键），同时，{SNO}还是关系变量LS和NLS的外键，引用关系变量S的键{SNO}，我之前并没有费心去澄清这些外键到底怎么划分。[1] 注意：在前几章中我们已经提过很多次了，实际上在关系变量LS中我们可以去掉属性CITY，因为它的值是不变的（总是伦敦），但在这里我不会这样做，以便保持例子的简单性。当然，如果我们去掉了这个属性，那么LS将不仅仅是S的限制这么简单了，而在本章中我想把我的精力主要放在限制相关的问题上面。

样本值如图 4.1 所示。

S

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

LS

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S4 | Clark | 20 | London |

NLS

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S5 | Adams | 30 | Athens |

图 4.1　关系变量 S、LS 和 NLS 的样本值

现在让我们来回顾一下“可交换性原则”，根据该原则，在基础关系变量和视图之间应该没有任何随机的和不必要的区别（也就是说，视图在用户角度审视，应该“看起来和感觉上”跟基础关系变量一样）。由这个原则导致的一个结果就是在这个例子中，关系变量 S、LS 和 NLS 的表现不应当取决于它们之中谁是基础关系变量，谁是视图。因此，在下一个明确说明之前，我们姑且假设它们都是基础关系变量。

```
VAR S BASE RELATION { ... } KEY { SNO } ;
```

[1] 在由我和 Hugh Darwen 合著的《Database Explorations: Essays on The Third Manifesto and Related Topics》（Trafford 出版，2010 年）这本书中有一篇论文名叫“Inclusion Dependencies and Foreign Keys”，根据这篇论文所阐述的观点来看，{SNO}在关系变量 S 中也是一个外键，引用关系变量 LS 和 NLS 交集为空的合并中的键{SNO}。

```
VAR LS  BASE RELATION { ... } KEY { SNO } ;
VAR NLS BASE RELATION { ... } KEY { SNO } ;
```

下面我们所重复的第 1 章的内容就是这 3 个关系变量的谓词（请注意对谓词来说，我更喜欢不正式的散文风格的定义）。

S：供应商 SNO 已经签约，名称为 SNAME，拥有状态 STATUS，并且位于城市 CITY。

LS：供应商 SNO 已经签约，名称为 SNAME，拥有状态 STATUS，并且位于城市 CITY（CITY 均是伦敦）。

NLS：供应商SNO已经签约，名称为SNAME，拥有状态STATUS，并且位于城市 CITY（CITY不是伦敦）。[2]

同样，如同我在第 1 章中所讲的，除了我们已经提到的最关键的约束外，所有这些关系变量都受制于其他一些约束条件。（在第 1 章中我并没有明确这些约束是什么，不过接下来我会逐条说明。）首先，下面两条约束条件显然是要遵守的。[3]

```
CONSTRAINT ... LS  = ( S WHERE CITY = 'London' ) ;
CONSTRAINT ... NLS = ( S WHERE CITY ≠  'London'  ) ;
```

接下来两条也是。

```
CONSTRAINT ... IS_EMPTY ( LS  WHERE CITY ≠  'London'  ) ;
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
```

通过前面所列出的这些约束条件其实我们可以推断下面几条也是其约束条件（尤其包含特定的外键约束，但是同样我不会在这里体现出来）。

```
CONSTRAINT ... S = UNION { LS , NLS } ;
CONSTRAINT ... DISJOINT { LS , NLS } ;
```

关于上面这两个额外约束中的第一个，我应当先解释一下 Tutorial D，至少是在本书中使用的版本，是同时支持特定的二元（中缀）和多元（前缀）关系操作的，特别是这里面还包含合并（也包括连接，这点我们将在下一章看到）。我们举例来说，LS 和 NLS 的合并

[2] 在第 2 章中，我曾经将下面这条当作经验法则：各关系变量的谓词应当交集为空，因此应当防止数组同时满足它们的情况出现。尤其请注意观察我在这里是如何把这条法则应用在 LS 和 NLS 的谓词上的。

[3] 约束条件是使用 Tutorial D 表达的，不过请注意，我为了简化表达而省略掉的约束名称在 Tutorial D 的实际使用中是需要写清楚的。

可以用 LS UNION NLS 或者（像上面约束中写的一样）UNION {LS, NLS}之中任意一种方式来表达。在本书中，这两种方式我都会用到，不过我会根据表述需要按照适用程度来选择。那么对于上面两个“额外”约束中的第二个，我应该解释一下 Tutorial D 的表达式 DISJOINT {r1,...,rn}：当且仅当变量 r1…rn 中没有任何两个变量具有相同的数组时，DISJOINT {r1,...,rn}返回“TRUE”。顺便说一句（你可能想起来第 1 章中曾提到过）这个约束同许多关键约束一样，都能够推导出下面这个约束。

```
CONSTRAINT ... DISJOINT { LS { SNO } , NLS { SNO } } ;
```

（“供应商编号不会同时出现在 LS 和 NLS 中”。）

重要：请注意观察前面所述的这些约束是如何共同作用来保证我早前提到的信息等价的。事实上，我们正好可以通过它们来比较一下使用 S 自己和综合使用 LS 和 NLS 这两种设计方式是如何映射到对方之中的。

作者注：当然，一个数据库如果包含全部这 3 个关系变量的话肯定会有一些冗余的部分。不过，上面列出的这几条约束（至少是包含多变量的这几条），可以通过一种正式的方式帮助我们来捕捉到那些冗余的部分。但是正如我所解释的那样（请参看《Database Design and Relational Theory》），只要这些冗余的部分被合理地管控，它们就不会造成任何麻烦。对于这点我将会在下一节“更多关于补偿性操作的内容”中做出简要的说明。

关于名词术语，上述 CONSTRAINT 声明中的一些，具体来说就是上述所有的以 <relation expression> = <relation expression>方式写成的布尔表达式，需要一个特定的“相等依赖条件”（必须随时满足）才能够正常起效。下面是定义。

- **定义：**相等依赖条件（简称 EQD）用 rx1=rx2 来表示，其中表达式 rx1 和 rx2 分别表示关系 r1 和 r2，并且 r1 和 r2 是相同类型。当且仅当 r1 和 r2 的关系体相等时，EQD 成立。

而相等依赖条件其实是另一种更加常见的现象的特殊情况，这种现象被称为“包含依赖条件”。

- **定义：**包含依赖条件（简称 IND）用 rx1 ⊆ rx2 来表示，其中表达式 rx1 和 rx2 分别表示关系 r1 和 r2，并且 r1 和 r2 是相同类型的。当且仅当 r1 的关系体是 r2 关系体的一个子集时，IND 成立。

对于后面这个定义，有下面几个重点需要注意。

- 外键约束是 IND 的一个特殊情况。例如，在关系变量 LS 中，{SNO}是一个外键，引用关系变量 S 的键{SNO}，因此 LS 对 S 来说满足 IND——LS 在 SNO 上的投影包含于 S 在 SNO 上的投影（用符号表示就是 LS{SNO} ⊆ S{SNO}）。
- 前面已经说过，EQD 也是 IND 的一个特殊情况。实际上，EQD r1 = r2 等价于两个 IND r1 ⊆ r2 和 r2 ⊆ r1 同时成立的情况。换句话说，EQD 成立是两个变量之间一对 IND“相互成立”时的情况。
- 虽然目前所说的与视图更新话题没有什么直接的联系，但是这一点我一定要提及,那就是任何一个使用 Tutorial D CONSTRAINT 来表达的约束实际上都可以通过一个 EQD 来表达（这里请读者自己做几个相关练习试试看）。

## 4.2 更多关于补偿性操作的内容

在第 1 章中，我们看到了补偿性操作是如何在“抛砖引玉”的例子中使用，从而确保在更新结束后所有的相应的约束还继续生效的。这里我们所说的补偿性操作（或规则）如下所列。

```
ON DELETE d FROM LS : DELETE d FROM S ;
ON DELETE d FROM NLS : DELETE d FROM S ;

ON DELETE d FROM S : DELETE ( d WHERE CITY = ‘London’ ) FROM LS ,
                     DELETE ( d WHERE CITY ≠ ‘London’ ) FROM NLS ;

ON INSERT i INTO LS : INSERT i INTO S ;
ON INSERT i INTO NLS : INSERT i INTO S ;

ON INSERT i INTO S : INSERT ( i WHERE CITY = ‘London’ ) INTO LS ,
                     INSERT ( i WHERE CITY ≠ ‘London’ ) INTO NLS ;
```

这些具体的规则在第 1 章中我们或多或少都有所解释，但是关于它们仍然还有许多值得说的内容，因此我们将用这一节来做个补充，具体如下。

- 首先，你可能已经注意到我在术语的使用上有点不那么精准。其实，我使用了词语“补偿性操作”来表达两层意思，（a）补偿性操作，都是在具体的更新操作完成之后才执行的，（b）在实际应用的语法中对这类操作的正式叫法（虽然我也用“规则”这个词语来表达后面这一点）。而在后面的章节，我会继续保留这种说法，希望这不会造成混淆。
- 例子中的这些规则都是在假设的前提下，在Tutorial D环境中，故意用比较冗长的方式来表达的。合在一起，它们可以体现后面几页里我将要对补偿性操作使

用的一种语法。[4] 请注意，d和i都是对这些规则有效的参数，分别代表在规则需要补偿的对象提出更新需求后相应要删除和插入的集合，在规则中使用DELETE d FROM R和INSERT i INTO R的形式表示。同时也请注意，正因为d和i各自的身份，导致它们的交集为空。

- 补偿性操作可以使用多种语法运用不同方式表达，但在语义上是等价的。例如，从 S 到 LS 和 NLS 的删除规则如下。

```
ON DELETE d FROM S : DELETE ( d WHERE CITY = 'London' ) FROM LS ,
                     DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ;
```

  也可以像下面这样来表示。

```
ON DELETE d FROM S : DELETE ( LS MATCHING d ) FROM LS ,
                     DELETE ( NLS MATCHING d ) FROM NLS ;
```

  甚至还可以用下面这种形式来表达如下。

```
ON DELETE d FROM S : DELETE d FROM LS ,
                     DELETE d FROM NLS ;
```

  当然，前提是“要删除一个不存在的数组”并不是一个错误。对于这一点如果我们要讲得极端一些，那就是我们总是可以单独只用关系赋值来表达这些规则。[5] 而为了教学的目的，我更倾向于使用INSERT和DELETE来更清晰地表达这些规则。但我必须要强调，在本书中我最主要的目的是保证所有的公式表达在逻辑上是正确的，而并不是在目前的阶段找出最简便的或者最有效率的公式表达方式。

- 这里解释一下前面的一点。我希望大家通过我在第 2 章的说明能明显地看出，我的目标总是要让理论正确，而不是去担心执行方面的问题（至少现在还不用担心）。而毫无疑问在实际的应用和执行过程中总会出现需要改进的问题。注意：在这里我还是想提一个这类问题，那就是只要 DELETE d FROM R 这个语句出现，d 就可以安全地被 d INTERSECT R 替代，同样的，只要 INSERT i INTO R 这个语句出现，i 都可以安全地被 i MINUS R 替代。为了便于说明和阅读，在接下来的文章中，我会假设这些替代总是发生，也因此 d MINUS R 和 i

---

[4] 这并不是说我不可救药地“嫁给了”这种语法（我希望没有），只是这种语法能够支持我在本书中所要表达的内容，仅此而已。

[5] 例如，下面这段：ON <any update> TO S : LS := S WHERE CITY = ‘London’, NLS := S WHERE CITY ≠ ‘London’。现在，你是不是觉得这种形式的规则表达式特别有吸引力？因为它看起来比我选择的用 INSERT 和 DELETE 逐条写清楚的方式要简单得多。然而，我的观点是这类所谓的“简单”基本都是假象。因为它很明显在实际应用中更难实行，尤其是对很多重要问题，诸如如何对“单纯的 INSERT”或“单纯的 DELETE”操作要求做出响应等。

INTERSECT R 总是都为空。换句话说（尽管前面的注释有说过这点），我会假设这类规则不会要求将一个已经存在的数组插入到变量，或将一个不存在的数组从变量中删除。

- 假设一个用户U可以看到所有 3 个关系变量S、LS和NLS，并且针对关系变量S执行了一个DELETE操作。那么，由于该关系变量上的删除规则生效，这个DELETE操作（一般情况下）将会同时从关系变量LS和NLS上删除相应的数组。**这条规则必须是对用户U清晰可见的**，否则的话用户U就会发现上面的情况违反了赋值原则（也是一般情况下）。具体来说就是如果用户U不知道这条规则，那么他就很可能看到数据库变量发生变化，也就是代表整个数据库的变量发生变化，而这并不是该用户的原始需求。同样的，该用户也会看到并没有明确要求过的更改发生在变量LS和/或NLS上。因此，下面两条一定要注意，（a）补偿性操作不能被当作“卧底”来对待，反而必须要让用户知道它的存在；（b）用户必须清楚地知道他们所要求的某一项更新实际上是一系列扩展操作的综合体，尤其是对那些特定的多重赋值来说。[6]

- 继续上面的一点，如果用户发起了一个更新需求，那么这个更新连同所有和它有关的补偿性操作必须作为一个整体，在语义上构成一个原子操作，具体来说就是一个多重赋值操作。因此，在这个多重赋值被整体执行完成之前，是没有一致性检查的。注意：正如第 2 章中所讲过的，通常情况下，对于组成多重赋值的每一个单独的赋值，我们都可以认为它们是同时发生的。不过对于我们现在所讲的这个例子而言，为了能够更好地服务于本书，我会假设更新部分的操作首先完成，补偿性操作然后完成（可能这里使用“在其之后”比“在其之上”更容易让大家理解）。但是请注意，这个假设并“不是”逻辑上的要求，只是这样假设能够简化相应规则的表达式（在通常情况下）。

- 补偿性操作本身也是一个更新操作，因此一般它的被执行可能导致特定的更进一步的补偿性操作发生，以此类推。我们举个例子，针对关系变量S的DELETE操作，从S到LS的级联删除将会引发从LS到S的级联删除规则被调用。当然这里所说的第二个级联删除对本范例是无效的，但是它确实给我们引出了一个有趣的问题，那就是通常情况下级联操作到底该在什么时候停止。我相信这是一个值得深入研究的问题。而为了更好地服务于本书，我将选择从实际角度出发来说明，那就是级联操作会在达到一个“固定点”的时候停止，当达到了这个

[6] 注意观察这类情况与下面情况的相似性。如果一个用户同时可以看到关系变量 S 和关系变量 SP，那么在 S 与 SP 存在级联删除规则的情况下，该用户被允许对关系变量 S 执行删除操作。

点时，数据库不再发生新的变化。[7]注意：假设U1 是用户提交的更新需求，对U1 的补偿性操作又导致了另一个更新需求U2 产生，而它又导致了另一个补偿性操作被执行（以此类推），这个时候U2 又成为了后面补偿性操作的“原始更新需求”，以此类推，递归循环。当然，执行所有的这些更新U1，U2……集合的时候，语义上必须还是构成一个原子操作。

- 最终，让我们来检验一下下面的情况：LS 和 NLS 是视图，而 S 是一个基表，但是请记得，目前为止，所有这 3 个关系变量都是基础变量。考虑到这种情况，我观察到诸如我们所讨论的这些补偿性操作正是关系变量之间“控制冗余”所需要的因素。换句话说就是在一般情况下，即使数据库根本不支持视图，补偿性操作也是存在的。注意：冗余如果存在的话应当是被“控制”的（并且用户了解这一点），但是为了确保一致性的“扩大化更新”任务却被系统管理，而不是被用户。（不可控的冗余会成为问题，而可控的冗余却不会。）控制冗余需要引入以下两点：（a）向数据库管理系统清晰明了地说明冗余的情况，（b）设置数据库管理系统负责进行必要的扩展更新。而对于我们手中这个例子来说，说明冗余的情况，就是要说明哪些多重变量约束是适用的。

- 前面的一点实际上是一个特殊情况，而一般情况已经在第 3 章中提到过，即补偿性操作无论如何都会存在从而保持特定的多变量约束能够起效。当然，在这里我假设我们并不想让用户自己负责编写出必要的多重赋值，因为每次他们执行一个更新，（他们写的）多重赋值都可能违反某条约束。我们换一种方式理解：根据定义来看，多变量约束通常意味着特定的更新在逻辑上必须是多重赋值。所以我们需要在下面两种情形中做出选择：要么（a）我们要求用户必须总是自己来编写出那些多重赋值，要么（b）我们找到一种方法让系统来做一部分工作，这样用户每次基本只写一个单独赋值就可以了。补偿性操作就是为了达成选项（b）而存在的。

- 有一位本书早期草稿的审阅者曾经问我，在关系变量 S、LS 和 NLS 的例子中，为什么补偿性操作都必须是级联的？具体来讲，那位审阅者指出了下面两点：（a）SQL 允许一种“引用操作 referential action”[原文如此]叫作 NO ACTION，这个操作可以代替级联，原理是如果某个更新会违反引用完整性约束，那么这个更新就会直接失败。因此他指出了第二点（b）为什么我们不能在自己讨论的例子中有一个类似的选项呢？下面针对这个疑问我有两个回应。第一，

[7] 我很抱歉在这里使用了逻辑性不那么强的词语“固定点”。这并不是我造的词。更重要的是我们要注意下面两个默认的假设：（a）“固定点”一定会达到，（b）这个点是独一无二的！让我再重复一次，我相信这方面的问题需要更多的研究。

如果提出类似的这个选项的话意味着我们可能要让用户总是自己来编写这些多重赋值，而不是让系统来承担一部分工作（请参看我前面一条着重记号的内容）。第二，当我们考虑到这个例子的真实对象，即例中的 S 是基础关系变量，而 LS 和 NLS 则是视图时，我们就会发现级联是“一定会”发生的事情。

- 在本节我还有最后一点要说，而且是很重要的一点，它是前一点的延续。请大家回忆一下第 2 章最后一节的这段话：

“关系变量”（至少是在数据库中的关系变量）并不是真正的变量，它们只是一些让人觉得很便利的虚构的变量，能够制造出这种假象，即数据库（或数据库变量）可以一点一点地、一个关系变量一个关系变量地被更新。我们进一步可以推断出，在某种意义上，“关系赋值”（不管是不是多重的）也同样是个方便的假象，具体来说就是它是一个虚拟操作，让我们感觉好像可以通过一系列针对这个数据库内的单独的关系变量的更新，来实现对数据库变量的更新。

但是我现在必须说：这个“虚拟的假象”并不是百分之百能够一直持续正常运作的。这是因为通常情况下在实际环境中进行更新，尤其是视图更新时，并不总是通过单一赋值就能实现（因此我们只能对变量逐个更新）。下面几章我会详细地介绍补偿性操作，目前来看，它们大多数时间都能很好地维持这些“虚拟的假象”正常运作，但是在某些情况下（当遇到时，我会明确地指出）它们可能会由于某些原因而产生出让人无法接受的效果。而在这些情况下（就是效果让人无法接受的情况下），我想除了让用户自己来编写出一个适用的多重赋值（其实就是一个适用的“数据库”赋值）以外，别无他法。

## 4.3 关于触发器

在这里我想先暂停一下说点别的，以防你们会产生另一个反对意见，即整个“补偿性操作”的想法不就是把 SQL 风格的触发器换了个名字搬出来么？换句话说就是我说了半天有什么是新东西吗？

显然，在这两个概念之间确实存在着一些相似的地方。如果（目前是很典型的情况）系统提供很少或者不直接提供针对补偿性操作的支持，我们甚至可以通过触发器来实施补偿性操作。[8] 但这里同时也存在着很多不同，下面列出一些。

[8] 一位审阅者曾就这一点做出过批注，如果我们在这里使用触发器的话要注意下面两点：（a）通常来讲这样使用触发器是非平凡的，（b）在某些情况下，如果系统不允许触发器被循环调用，那么是不可能这样使用触发器的。

- 触发器可以并且经常引入过程代码。事实上，我查阅了一些关于这个问题的教科书，既包括针对 SQL 商业产品的，也包括针对 SQL 标准的，发现比较通用的假设是触发器总是引入过程代码。对比来说，在本书中我所介绍的补偿性操作则更多地可以使用自然语言来表述。（当然，对于“可表述性”和“过程”这两个词并没有正式的定义，所以这一点表达的可能没想象中有那么精确。但在这里我想说明的是，通常情况下我所提出的补偿性操作确实像 Tutorial D 一样是可表述的。除此之外，这些操作是可表述的这个事实也意味着数据库管理系统可以“理解”这些操作要做什么并且可以通过正式的渠道把它们合理化。）

- 根据上面说的展开来看，触发器通常可以做任何事，换句话说就是，它们可以引入任意复杂度的程序，而补偿性操作最复杂的程度，也不过是综合执行一些关系型 INSERT 和 DELETE 操作。

- 对于触发器来说，系统不可能自己决定下一步执行哪些操作（如果系统有这个能力，那么触发器的存在就显得没那么必要了）。也就是说，触发器是用户自己来定义的。而我们反观补偿性操作，正如我在第 1 章所说的系统应当能够自己确定需要执行哪些操作，换言之，这些操作应该是由系统定义的，而不是用户。

- 通常来讲，触发器的操作细节，甚至是否存在这点，对于用户来说都是被隐藏起来的。由此导致的结果是从用户角度来看他们很可能认为使用触发器就会导致违反“赋值原则”。这也是相对于补偿性操作的情况而言的。

- 尤其是在 SQL 环境中，触发器能够并且经常违反集合级别的关系模型特性。正如我在第 1 章所说的，根据定义来说关系更新是集合级别的，并且它们一定不能被当作是一系列单独数组级别更新（在 SQL 环境中是行级别更新）的集合，尤其是补偿性操作必须在用户要求的更新完全执行完毕之后才能够开始执行，而完整性约束则是必须要在更新“和”任何相关的补偿性操作全部完成之后才开始检验。目前来看 SQL 支持一种叫作“行级别”的触发器，而它则是完全违反关系规则的。

- 最后（通常与触发器不同），由于“可交换原则”的关系，我所说的关于视图更新的这些补偿性操作是逻辑上必需的。[9]

---

[9] 至少在我们有信息等价的情况下，它们就是逻辑上必需的。

## 4.4 关于显式更新操作

假设在关系变量 S 上，我们有下列显式 UPDATE 操作。

```
UPDATE S WHERE CITY = 'Paris' : { STATUS := 30 } ;
```

在《SQL and Relational Theory》一书中我们解释过，这个操作实际上被定义为下列关系赋值的简化写法。

```
S := ( S WHERE CITY ≠ 'Paris' )
       UNION
      ( EXTEND S WHERE CITY = 'Paris' : { STATUS := 30 } ) ;
```

换句话说，这个更新整体的效果（至少从概念上讲）是首先删除所有位于巴黎的供应商，然后把它们的状态值设为 30 再重新插入进来。不过请注意观察，这里有 2 个集合——位于巴黎的供应商，和状态值为 30 且位于巴黎的供应商，可能交集不为空，因为有些位于巴黎的供应商在更新完成之前的状态值就已经是 30 了（我们以供应商S3 为例，取它的值作为参考值）。因此，如果我们想要确保删除集合d和插入集合i确实交集为空——在这里我们能够确定 [10]，那么我们就必须将原来的UPDATE扩展式重新修改为如下形式。

```
S := ( S WHERE CITY ≠ 'Paris' OR STATUS = 30 )
       UNION
      ( EXTEND S WHERE CITY = 'Paris' AND STATUS ≠ 30 ) :
                                           { STATUS := 30 } ) ;
```

从这个扩展式中我们得出 d=状态值不为 30 的位于巴黎的供应商，以及 i=状态值为 30 的位于巴黎的供应商,并且这两个集合的交集为空，满足要求。

到目前为止，在刚刚的这个例子中我故意忽略了补偿性操作，因为它们暂时跟我要表述的观点还没什么关系。不过现在还是让我们来看一看这些针对显式 UPDATE 操作有关的操作能带来什么样的效果吧。但是请一定记住，目前我还是假设所有涉及的关系变量都是基础关系变量，暂时不包含视图。注意：接下来的讨论是部分（但也仅仅是部分）重复第 1 章的一些内容。

首先是第一个例子，我们设下列 UPDATE 操作在关系变量 NLS 上执行。

```
UPDATE NLS WHERE SNO = 'S2' : { CITY := 'Oslo' } ;
```

为了能够更简洁地表述，让我们假设，我们已经知道供应商 S2 所在的城市并非奥

---

[10] 说得明确一点，我们当然不想要这种先删掉一些东西，再插入一些东西，相当于取消之前删除的无用功。

斯陆。那么接下来将会发生的情况如下。

1．代表供应商 S2 的已存在的数组被从关系变量 NLS 中删除，然后一个 CITY 值是 Oslo 的代表该供应商的新数组，被插入到相同的关系变量中。注意：为了简便我在这里（并且在本书中所有需要这样做的例子中）假设我们可以单就删除和插入的这些数组，而不是整个关系进行讨论。不过从技术上讲，你应当明白，这类讨论针对的是数组的“集合”，只是这些集合的势刚好为 1（也就是集合的元素只有一个）。

2．由于从 NLS 到 S 的级联删除规则起效，代表供应商 S2 的已存在的数组也被从关系变量 S 中删除，而由于从 NLS 到 S 的级联插入规则起效，一个 CITY 值是 Oslo 的代表该供应商的新数组，也被插入到关系变量 S 中。

请一定多加注意，前面这些 DELETE 和 INSERT 操作都是作为同一个原子操作（当然是一个多重赋值）的一部分来执行的。尤其是，一致性检查要在所有的 DELETE 和 INSERT 操作结束后才会发生。

接下来是第二个例子，设下列 UPDATE 在关系变量 S 上执行。

```
UPDATE S WHERE SNO = ‘S2’ : { CITY := ‘London’ } ;
```

为了便于理解我们假设已经知道供应商 S2 的所在城市并非伦敦。那么将出现如下情况。

1．当前存在的供应商 S2 的数组被从关系变量 S 中删除，然后一个 CITY 值为 London 的代表该供应商的新数组被插入到相同的关系变量中。

2．由于从 S 到 NLS 的级联删除规则起效，代表供应商 S2 的已存在的数组也被从关系变量 NLS 中删除，而由于从 S 到 LS 的级联插入规则起效，一个 CITY 值是 London 的代表该供应商的新数组，也被插入到关系变量 LS 中。换句话说就是代表供应商 S2 的数组从关系变量 NLS“迁移”到关系变量 LS 中——当然这种说法“相当地”随意。

最后再举一个例子，假设上述的 UPDATE 已经被指向关系变量 NLS，而不是关系变量 S。

```
UPDATE NLS WHERE SNO = ‘S2’ : { CITY := ‘London’ } ;
```

那么就会出现下面的情况。

1．当前存在的代表供应商 S2 的数组被从关系变量 NLS 中删除。

2．系统试图向 NLS 插入一个 CITY 值为 London 的代表供应商 S2 的新数组。但是这种尝试失败了，因为它违反了一条NLS的约束条件，那就是在该关系变量中CITY值永远不能为 London。于是整个更新失败，之前的一步（也就是从 NLS 中删除代表供应商 S2 的原始数组）被取消，最终效果就是数据库维持原样没有发生改变。

## 4.5 供应商与设备供应

本节并非是主线内容，并且在你第一次阅读的时候它很有可能被跳过。一方面，本节中所使用的例子从性质上讲与目前为止本章所考虑的（也就是伦敦的供应商对比非伦敦的供应商）并不相同，它同供应商与零部件数据库中的关系变量 S 和 SP 有关，并且毫无疑问其中的一个关系变量是另一个的限制条件，同样确定的是其中并没有一个变量是另一个的视图。另一方面，虽然总的来看我们将要讨论的问题与更新是有关的，但具体来说似乎与视图更新（或者冗余控制）没有很大关系。由于以上这些原因，接下来我们所讨论的内容其实并不属于本章，反而它更应该被放到第 2 章或者第 3 章中。但是它却又必须仰仗一些特定的素材，特别是在上面一节中所讨论到的显式 UPDATE 操作内容，而这些却又没有被涵盖于第 2 章和第 3 章中（放在第 2 章和第 3 章中确实不太合理），因此我决定将它放在这里讲。

让我们考虑一下在供应商与零部件数据库上从设备供应关系变量SP到供应商关系变量S的外键约束。针对这个讨论，我们假设要确定一个跟这个外键有关的级联删除，于是：[11]

```
ON DELETE d FROM S : DELETE ( SP MATCHING d ) FROM SP ;
```

作者注：既然关系变量 SP 并不是由关系变量 S 来定义的，尤其是，SP 并非 S 的视图，那么毫无疑问数据库管理系统会单独从关系变量定义的角度来判断哪个（些）规则是适用的。而对于数据库管理员或者某些被授权的用户来说，他们则只能想办法将这些规则逐条弄清楚。

下面我们来看这个针对关系变量 S 的 UPDATE 操作。

[11] 在 SQL 中，这类规则应该能够更加简单（?）地使用 ON DELETE CASCADE 这个格式来定义，并且被稍微有点不合逻辑地作为对表 SP 的 CREATE TABLE 声明的一部分。（这里我说“不合逻辑”是因为删除操作其实是对表 S 而非对表 SP 进行的操作。）所以我更倾向于使用具备以下两点特征的定义：（a）使用我自己的故意有点冗长的语法，以及（b）可以作为一个独立而正确的声明存在。

```
UPDATE S WHERE SNO = ‘S1’ : { STATUS := 10 } ;
```

为了便于理解我们假设已经知道供应商 S1 的状态值并不是 10。那么会出现如下情形。

1．供应商 S1 的“供应商”数组已经从关系变量 S 中删除，并且一个表示该供应商的状态值为 10 的新数组被插入到相同的关系变量中。

2．由于从 S 到 SP 的级联删除规则，供应商 S1 的那些“设备供应”数组也已经从关系变量 SP 中删除。但是，并没有从 S 到 SP 的级联插入规则存在，因此并不会有新的设备供应数组插入到 SP 中，这与上面讲的会有代表供应商 S1 的新数组插入到 S 中不同。最后结果：供应商 S1 的设备供应数组丢失！

在本例中很关键的一点是我们要正确理解，不仅这里没有从 S 到 SP 的级联插入规则，而且是“不能够”有，因为这种规则根本不合理（对吧）。

但是前面所发生的情况也同样是不可接受的。我是说，我们必须能够在进行类似改变状态这类操作的时候，保证不丢失该供应商的“设备供应”信息。因此很明显，我们需要对例子中的这类 UPDATE 操作做一些调整。但是我们应当做些什么呢？

我认为，从总体上来说这个问题的关键在于更新操作同时引入了一个删除集合与一个插入集合，也正因为如此，补偿性操作在一般情况下所需要做的并不是仅仅分别执行删除或插入操作，而是需要对这两个操作进行合并执行。下面我们举一个对“供应商和设备供应”比较恰当的合并规则为例（注意我们已经介绍过的名字 t1 和 t2）。

```
ON DELETE d FROM S , INSERT i INTO S :
   WITH ( t1 := SP MATCHING d , t2 := SP MATCHING i ) :
   DELETE t1 FROM SP , INSERT t2 INTO SP ;
```

如果能够将补偿性公式化而不用带着那些介绍过的名字，那将是非常具有吸引力的，于是有：DELETE (SP MATCHING d) FROM SP, INSERT (SP MATCHING i) INTO SP。但是这种形式的公式有个问题，那就是“SP”这个符号在子表达式 SP MATCHING i（这个子表达式是 INSERT 公式的一部分）中代表的内容是执行了 DELETE 部分**之后**的关系变量 SP 的值，这个问题我们在附录 A 中也会做出解释。

另外，请注意这一点，没有任何补偿性操作是为关系变量 SP 定义的。当然这并不是说我们不能对 SP 做更新，只不过我们对 SP 做了更新它们也无法被执行。

现在，让我们再来回顾一下之前讨论的 UPDATE 的例子（为了方便查看，我在这

里重复写一下）。

```
UPDATE S WHERE SNO = 'S1' : { STATUS := 10 } ;
```

给我们常用的几个样本赋值，就可以得到下面的结果（让我们对数组再次用一个简化的标记，对删除集合与插入集合同样适用）。

```
d  = the S tuple (S1,Smith,20,London)
i  = the S tuple (S1,Smith,10,London)
t1 = existing SP tuples for supplier S1
t2 = existing SP tuples for supplier S1(same as t1)
```

于是我们可以看到，最终效果（大体来说）就是更新 S1 的供应商数组，把状态值从 20 变为 10，同时保证关系变量 SP 没有变化。

我会把它作为一个例子用来确认对于关系变量 S，在前面修改的规则之下，显式的 INSERT 和 DELETE 操作会如预期般持续有效。但对于下面这个 UPDATE 例子来说又会如何呢？

```
UPDATE S WHERE SNO = 'S1' : { SNO := 'S9' } ;
```

注意这个例子与我在第 2 章中引用的“键 UPDATE”操作一致。这次我们得到：

```
d  = the S tuple (S1,Smith,20,London)
i  = the S tuple (S9,Smith,20,London)
t1 = existing SP tuples for supplier S1
t2 = empty
```

如你所见，即使在修改规则之后，这个例子中依然有数组丢失，具体来说就是供应商 S1 的设备供应数组直接“消失”了，而“没有”被供应商 S9 的数组代替。

目前，这类例子可以并且已经被（至少是我）用来支持我关于显式“级联 UPDATE 规则”效果如何的争论中的观点，我们真正需要的是一个“级联 UPDATE 规则”吗？它的效果如何呢？在我们所讨论的例子中，这个规则将会在对关系变量 S 的 SNO 进行 UPDATE 操作时，生成一个对关系变量 SP 的 SNO 起效的级联更新（当然这是“非常”笼统地讲）。但在第 2 章中我就已经拒绝了这个显式 UPDATE 规则，并给出了我拒绝的理由。现在我再次说明一下，我们真正想要的是让补偿性操作由适当的删除集合与插入集合来驱动，而不是由随机选择的语法来驱动，更何况那些更新需求还是碰巧被公式化的。因此我直接放弃了这种方式。于是乎，如果想把在关系变量 S 中的某个数组的键值变化复制到对应的关系变量 SP 的数组中的话，那么我别无选择，只能明确地手写出必要的多重赋值语句（由此，如本章前面所提到过的，要有效地写出必要的“数据库”赋值），如下面的例子。

```
UPDATE S WHERE  SNO = ‘S1’ : { SNO := ‘S9’ } ,
UPDATE SP WHERE SNO = ‘S1’ : { SNO := ‘S9’ } ;
```

那么会发生如下几种情况。

1．关于供应商 S1 的“供应商”数组被从关系变量 S 中删除，另一个将 SNO 值从 S1 变为 S9 的与它完全对等的新数组，被插入到关系变量 S 中。同样的，所有对于供应商 S1 的“设备供应”数组也都被从关系变量 SP 中删除，一系列与“老”数组完全对等并且将 SNO 值从 S1 变为 S9 的新数组，则被插入到关系变量 SP 中。

2．从 S 到 SP 的级联删除规则被调用，并且对所有关系变量 SP 中 SNO 值为 S1 的“设备供应”数组发出了删除需求。但是，这个需求并无响应，因为相应的数组已经被用户通过显式需求的方式删掉了。最终效果：对于将供应商编号从 S1 变为 S9 的操作，已经在关系变量 S 和 SP 中执行了适当的更新。

注意，由于上面所说的情况，对于从 S 到 SP 的级联删除规则在这个例子中并没有实际的意义。当然，这个情况并不是说该规则毫无用处，它对于 DELETE 操作来说还是有用的，只是它对于显式的 UPDATE 操作来说并没有太多意义，正如上面例子所示。

无论如何，在整个讨论中最重要的信息是如下这条：“一般来讲，补偿性操作并不是分别执行删除或插入操作这么简单，而是将两种操作整合起来一同实施。”当然，对于每一个具体的更新来说，不论删除集合 d 或是插入集合 i 都有可能为空（当 d 为空时，规则会简化为一个简单的插入规则；而当 i 为空时，规则就会变为一个简单的删除规则）。

# 4.6　再谈抛砖引玉

现在让我们回到关系变量 S、LS 和 NLS。接下来，让我们继续假设这些关系变量都是基础关系变量（依然不是视图）。不过，现在我要提醒你的是至少从概念上讲，我们可以认为关系变量 S 自己建立了一个数据库 DB1，关系变量 LS 和 NLS 一起建立了一个数据库 DB2，而 DB1 和 DB2 是信息等价的。假设有位用户只能看到数据库 DB2，那么这位用户：

1．他知道相应的谓词。

LS：供应商 SNO 已经签约，名字为 SNAME，拥有状态 STATUS，并且位于城市 CITY（而 CITY 是伦敦）。

NLS：供应商 SNO 已经签约，名字为 SNAME，拥有状态 STATUS，并且位于城市 CITY（而 CITY 不是伦敦）。

2．他知道所有对应的类型信息，知道{SNO}是每一个关系变量的键，并且知道下列约束。

```
CONSTRAINT ... IS_EMPTY ( LS  WHERE CITY ≠ 'London' ) ;
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
CONSTRAINT ... DISJOINT { LS { SNO } , NLS { SNO } } ;
```

这里需要注意的是这些约束仅仅针对关系变量 LS 和 NLS（该用户甚至不知道关系变量 S 的存在）。

3．他并“不”知道任何补偿性操作的存在（因为本例中所有的补偿性操作对关系变量 S 都存在某些引用）。

那么我希望你从这个用户的角度能够看到，所有的更新，即所有的 INSERT、DELETE、UPDATE，再广义一点说所有的关系赋值操作，都能够如预期般运行。

### 信息隐藏

那么如果说一个用户只能看到关系变量 NLS 的话，会怎么样呢？这类用户知道相应的谓词（参看上文），知道相应的类型信息，也知道{SNO}是关系变量的键，并且还知道下列约束。

```
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
```

但是很明显，这类用户并不被允许向关系变量 NLS 中插入数组，也不能在关系变量内更新供应商编号，因为这类操作可能会违反某些（有必要）对该用户隐藏的特定约束。换句话说，就是这类用户只能看到系统的一部分。正如我在第 1 章中所讲的，规则至上！如果要更准确一点来阐述这个事情，那就是这里没有信息等价。这里有与关系变量 S 有关的清晰的定义描述，但是与 NLS 无关，因此如果分开来看，只有在关系变量 S 上才有可执行的操作，而在 NLS 上则没有对应的操作。

即便如此，我还是需要进行更详细一些的说明……首先，当我说某些用户“不被允许”对某些关系变量执行某些操作时，我的意思是该用户不应当获得相关授权。换句话说就是我假设数据库系统管理员，或其他具有相应授权的用户，会适当地使用授权子系统来保证那些没什么意义的操作不能被发起或尝试执行。

其次，重复一下我在第 1 章里讲过的，我假设“有可能”无论怎样都允许用户来运

行这些操作，只要他或她准备好接受在某些操作被系统拒绝时，只能收到类似“系统就是这么规定的”的错误信息，而没有具体说明的情况。关键在于：一般来说，这类操作会成功还是会失败不仅取决于那些被隐藏的约束，而且取决于被隐藏的部分数据库目前的状况。让我们来看一个具体的例子，给定一些普通的样本值，将数组（S1, Smith, 20, Paris）插入 NLS 的操作会失败，而将数组（S6, Smith, 20, Paris）插入 NLS 就会成功。那么从用户角度来看的话，这类操作的运行状况则是既不可预测，又（当失败时）无法解释的。这样一种情况对我而言是毫无吸引力的，因此我并不推荐系统支持这种“可能性”。

最后，我已经说过实际上这个用户被允许更新（比如）在关系变量 NLS 上的供应商状态值，但不能向该关系变量插入数组。那么，你可能会奇怪这种情况很不合理——难道不是应当引入某些约束，让特定的 DELETE/INSERT 整合后可以简短表达为 UPDATE 操作吗？后面这一点正是关键。对于每一个作为（成功执行的）显式 UPDATE 操作一部分的隐式 INSERT 操作，总会有一个能够保证该操作不违反约束（包含那些对用户隐藏的约束）的隐式 DELETE 操作伴随产生。因此，用户被允许执行一些特定的显式 UPDATE 操作并不意味着他或她已经被默许执行显式 INSERT 操作。

与上面三段类似的情况后面还有很多，本质上来说就是任何我指出某些用户“不被允许”执行某些操作。每次遇到这种情况我都会不厌其烦地重复，而不是在这几个段落说说就完了。

## 4.7 概括总结

在这一节中我们会把所有的东西都归纳在一起来讲，以便结合所有的关系变量定义、CONSTRAINT 声明，以及补偿性操作来看看抛砖引玉的例子。但是，由于前面一章我所提出的争议，也就是我认为用语法来区别视图和基础关系变量的定义是错误的这个问题，在这里我要简化关系变量的定义，仅此一次，是为了省略掉上述区别以便大家阅读。

```
VAR S ..... RELATION { ... } KEY { SNO } ;
VAR LS .... RELATION { ... } KEY { SNO } ;
VAR NLS ... RELATION { ... } KEY { SNO } ;

CONSTRAINT ... LS =  ( S WHERE CITY = ‘London’ ) ;
CONSTRAINT ... NLS = ( S WHERE CITY ≠ ‘London’ ) ;
CONSTRAINT ... IS_EMPTY ( LS  WHERE CITY ≠ ‘London’ ) ;
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = ‘London’ ) ;
CONSTRAINT ... S = UNION { LS , NLS } ;
CONSTRAINT ... DISJOINT { LS { SNO } , NLS { SNO } } ;
```

```
ON DELETE d FROM S , INSERT i INTO S :
   DELETE ( d WHERE CITY = 'London' ) FROM LS ,
   DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ,
   INSERT ( i WHERE CITY = 'London' ) INTO LS ,
   INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS ;
ON DELETE d FROM LS , INSERT i INTO LS :
   DELETE d FROM S , INSERT i INTO S ;

ON DELETE d FROM NLS , INSERT i INTO NLS :
   DELETE d FROM S , INSERT i INTO S ;
```

事实上，我们应当进一步考虑是不是能将不同种类补偿性操作的规则整合在一起。例如，最后两部分可以整合为如下语句。

```
ON DELETE d1 FROM LS , INSERT i1 INTO LS ,
   DELETE d2 FROM NLS , INSERT i2 INTO NLS :
   DELETE d1 FROM S , DELETE d2 FROM S ,
   INSERT i1 INTO S , INSERT i2 INTO S ;
```

无论如何，多重赋值都可以意会为我们可以将若干不同的更新整合到一个符合逻辑的原子操作中，那么对于补偿性操作为什么不能这样做呢？实际上，对整个数据库实现单一整合规则这点从理论上讲是可行的，虽然从人为因素角度讲可能没什么吸引力。如果实现的话，这还要感谢 Hugh Darwen 的观测，我们将拥有唯一的变量（整个数据库）、唯一的规则（对所有补偿性操作的整合），以及唯一的约束（全局数据库约束）。

# 4.8 最后一点

我们终于讲到了我最主要的观点：**如果一部分或所有的关系变量都是视图，那么之前我所说的所有观点都适用并且保持不变**。例如，我们假设最初设置的时候就将 S 设置为基础关系变量，而 LS 和 NLS 则都是视图。于是有：

1. 前一节中的 6 个 CONSTRAINT 声明所规定的约束条件会被自动执行，正是因为 LS 和 NLS 都是 S 的视图。也就是说，对任何关系变量的任何一个更新操作都不可能违反这些约束。

2. 从S到LS和NLS的补偿性操作也会被自动执行，同样也是因为LS和NLS都是S的视图。也就是说，对S的更新操作会被自动反应到相应的对LS或NLS或者二者皆有的操作中。因此从S到LS和NLS的补偿性操作实际上是可以被“推导”出来的，它们是从相应的视图定义[12]中被推导出来的（说到“视图定义”，其实我指的是从底层

[12] 更准确点说应该是约束是从视图定义中推导而来的，而补偿性操作则是由约束中推导而来的。

基础关系变量到视图的“对应方式”，换句话说就是相应的“视图定义表达式”)。

请特别留意观察，例如，对 S 中某些供应商的编号进行变化，我们姑且说从 S1 变到 S9，这将导致 LS 和 NLS 中任何适当的地方都发生级联变化。与之对比而言，让我们回忆一下“供应商与设备供应”那一节，这种级联变化并不会在 S 和 SP 中发生。不过那是因为对于 S 和 SP 的情况与这里不同：对于 S、LS 和 NLS，我们只是讨论从用户的角度来看的可控冗余的问题，而对于 S 和 SP 来讲则不是这样。（事实上，让我们再就这点引申一下，我所表述的关于视图更新的宏观概念，是让用户看起来就是一个可控冗余的问题，前提是这个用户既能够看到这些视图，又能够看到定义这些视图的关系变量。更具体一点就是在跟用户谈到补偿性操作的时候我们不需要告诉用户有哪些是基础关系变量，而有哪些是视图。）

3．从 LS 和 NLS 到 S 的补偿性操作，这些是视图更新构想的产物，也会自动执行，同样也是因为 LS 和 NLS 是 S 的视图。也就是说，对 LS 和 NLS 的更新才是“真正的”对底层关系变量 S 的更新，并且这些更新在 S 中自动可见，如同在 LS 或 NLS 中一样。

4．我们假设一个用户只能看到视图 LS 和 NLS。那么我希望你能看到如下结果：这个用户可以对这些视图执行任何操作就好像它们本身就是基础关系变量一样。事实上，如同本章早先的一节“再谈抛砖引玉”中讲过的，它们在这里就是基础关系变量。当然它们是该例子的对象。

请特别注意，举个例子，尝试改变LS中供应商所在的城市并不会导致LS中的数组从LS“迁移”到NLS中，反而，这个尝试会由于违反**黄金法则**而失败。然而，如果按照传统的特殊方式来尝试视图更新[13]，那么这类迁移则很可能是预期会发生，并且可能已经出现的。

5．另一方面来讲，如果我们说一个用户只能看到视图 NLS，那么显然该用户所能执行的操作会被限制，就如同当 3 个变量都是基础关系变量时，该用户只能看到基础关系变量 NLS 一样（同样，也请参看“再谈抛砖引玉”这一节）。这个用户会被允许对该变量删除数组，和/或更新除 SNO 以外的其他属性，但就只有这些权限了。再次强调，规则至上。

[13] 这类方式可参考诸如 Don Chamberlin 撰写的《Using the New DB2: IBM’s Object-Relational Database System》（Morgan Kaufmann，1996）等书籍。

关于这最后一点，可能有人会想，如果一个用户只能看到视图 NLS，而其对该变量所能执行的操作会被限制，这一点会违反“可交换性原则”。但显然这点并不违反该原则，正如我们已经看到过的，如果 NLS 是基础关系变量的话，同样的限制也一样存在。

## 4.9 重叠限制

让我们再多看一点复杂一些的关于限制的例子。假设我们定义针对供应商关系变量 S、NLS（“non London suppliers”）以及 NPS（“non Paris suppliers”）的两个约束视图。这两个视图都拥有键{SNO}。样本值如图 4.2 所示。下面是这两个视图的谓词。

NLS：供应商 SNO 已经签约，名称为 SNAME，拥有状态 STATUS，并且位于城市 CITY（并且 CITY 不是伦敦）。

NPS：供应商 SNO 已经签约，名称为 SNAME，拥有状态 STATUS，并且位于城市 CITY（并且 CITY 不是巴黎）。

S

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

NLS

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S5 | Adams | 30 | Athens |

NPS

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

图 4.2 关系变量 S、NLS 和 NPS 的样本值

下面的约束（实际上是EQD）显然有效 [14]。

```
CONSTRAINT ... NLS = ( S WHERE CITY ≠ ‘London’ ) ;
CONSTRAINT ... NPS = ( S WHERE CITY ≠ ‘Paris’ ) ;
```

下面这个约束（另一个 EQD）也同样有效。

[14] 请注意观察，我们再次遇到了信息等价起效的情况（尤其是，关系变量 S 等价于关系变量 NLS 和 NPS 的合并，尽管这里的合并并非交集为空）。另外请注意，NLS 和 NPS 的谓词其实违反了第 2 章（一个注脚）所介绍的一项原则，大意是说关系变量 R1 和 R2 的谓词最好如此从而可以防止同一个数组同时满足两种谓词的可能性。

```
CONSTRAINT ... ( NLS WHERE CITY ≠ 'Paris' ) =
                         ( NPS WHERE CITY ≠ 'London' ) ;
```

正如最后这条约束所示，视图 NLS 和 NPS 有部分是重叠的，在这种情况下，通常来讲会有特定的数组同时出现在两个视图中。实际上，这些数组必须同时出现在两个视图中，从而保证上面的约束总是被满足。于是我们有了如下的这一点，设数组 t 是出现在关系变量 S 中的数组，当且仅当数组 t 的 CITY 值既不是伦敦，也不是巴黎的时候，数组 t 会同时出现在 NLS 和 NPS 中。（我顺便提一句，正因为这种情形，关系变量 NLS 和 NPS 一起违反了“正交设计原则 The Principle of Orthogonal Design”。想获得关于这一点的更详细解释，请参看《Database Design and Relational Theory》。）

由此可以判断，对NLS和NPS中任何一个变量的特定更新都一定会级联到另一个上，尽管是间接的方式。例如，下面是S、NLS和NPS的插入规则[15]。

```
ON INSERT i INTO S :
   INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS ,
   INSERT ( i WHERE CITY ≠ 'Paris' ) INTO NPS ;

ON INSERT i INTO NLS : INSERT i INTO S ;

ON INSERT i INTO NPS : INSERT i INTO S ;
```

现在我们来看看，如果向 NLS 中插入数组（S6，Lopez，30，Madrid）会出现什么情况。首先，这个插入会从 NLS 级联到 S，而相同的数组也因此会被插入到关系变量 S 中；接下来，插入会从 S 级联到 NPS，同样这个数组也会由此被插入到关系变量 NPS 中，所以从效果上讲，整个插入是从 NLS 级联到 NPS。

下面我们再来看看删除规则（同时也请大家注意观察，这里也有可能会有使一个操作在 NLS 和 NPS 之间，通过间接方式相互级联的情况出现）：

```
ON DELETE d FROM S :
   DELETE ( d WHERE CITY ≠ 'London' )FROM NLS ,
   DELETE ( d WHERE CITY ≠ 'Paris' )FROM NPS ;

ON DELETE d FROM NLS : DELETE d FROM S ;

ON DELETE d FROM NPS : DELETE d FROM S ;
```

因此如果一个用户只能看到 NLS 和 NPS，但看不到 S 的话，有时该用户会观察到 NLS 和 NPS 二者之间相互级联的情况。但当然我们很希望该用户能够将 NLS 和 NPS 当作基础关系变量来看。因此换句话说，即使它们真地就是基础关系变量，特

[15] 虽然我说过在实际应用中插入和删除规则可以整合在一起，但接下来大部分时间里，我还是会分开展示它们以便使阐述更清晰，更便于理解。

定约束的存在也能说明存在下面情况：在某些情况下，即使在基础关系变量之间，特定的级联在逻辑上也是应该出现的。实际上，关系变量 NLS 和 NPS 受制于一个特定的冗余，并且我已经讲过，当这个冗余存在的时候，它应当是被控制的，而这正是级联更新的功能所在，即将更新扩大化用来控制冗余。下面是相应的规则。

```
ON INSERT i INTO NLS :
   INSERT ( i WHERE CITY ≠ 'Paris' ) INTO NPS ;
ON INSERT i INTO NPS :
   INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS;

ON DELETE d FROM NLS :
   DELETE ( d WHERE CITY ≠ 'Paris' ) FROM NPS ;

ON DELETE d FROM NPS :
   DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ;
```

这些规则是经由对 S、NLS 和 NPS 已经存在的规则进行整合之后推导而来的，并且将对 S 的引用相应地替换为对 NLS 和/或 NPS 的引用。

那么关于信息隐藏呢？举个例子，比如对那些只能看到视图 NLS 的用户呢？我将把对这种可能性细节的考虑留做一个练习。

## 4.10 小结

我还有几个观点需要说明。第一点是关于专有名词的一个小问题。我已经谈了“限制视图”和“通过限制进行更新”，我也曾经讲过，例如视图 LS 是基础关系变量 S 的一个限制条件。但是限制是一个关系型代数的操作，因此根据定义来说，它并不真正作用于关系变量，而是对关系本身起效。那么我们探讨了那么多对于关系变量而不是对于关系本身的限制、投影或者连接（等）又有什么意义呢？其实，我在其他著作上已经写过相关的答案（请参考，《The Relational Database Dictionary, Extended Edition》，Apress，2008）。

根据定义，运算符投影、连接以及类似的操作是针对关系值适用的。当然，对于那些刚好是关系变量值的关系值也同样适用。因此我们来讨论下面的情况就变得很合理，例如，在当前情景下有一个关系值与关系变量 S 的当前值相等，该关系投影到属性{CITY,STATUS}上会产生另一个关系，这另一个关系就是关系变量 S 在属性{CITY,STATUS}上的投影（关系变量 S 在属性{CITY,STATUS}上的投影，其实就是由将与关系变量 S 当前值相等的关系投影到那些属性上所得来的关系）。而在某些教科书中（比如最标准的那种），面对稍有不同的情景，使用“关系变量 S 在属性{CITY,STATUS}上的投影 the projection of relvar S on attributes {CITY,STATUS}”这

种表达就会很方便。具体来说，我们可以看到不严谨，但是非常方便的是比如某个"关系变量"CT 是关系变量 S 在属性{CITY,STATUS}上的投影，更精准一点的意思是说，关系变量 CT 的值总是关系变量 S 在该命题提出时刻的值在这些属性上的投影。因此某种意义上讲，我们可以讨论关系变量的本质投影，而不是只能说关系变量当前值的投影。类似的解释适用于所有关系操作。

换句话说，我们经常讲（例如）限制关系变量一类的说法，尤其是我们在这里也会这样说。这种说法有点不那么合适，不是说有多么谬误，但至少是有些简略了。

我要说的第二点是除开别的不讲，我已经假设过 S 是基础关系变量，而 LS 和 NLS 都是 S 的（限制）视图这种情况。为了更全面地考虑各种状况，我应当同样也考虑反过来的情况，也就是 LS 和 NLS 是基础关系变量，而 S 是它们的视图这种情形。但是在这种情况下，S 就应当是个合并视图，因此我要把这种情况的讨论放到第 10 章来讲，在那一章我们将会整体讨论合并视图的主题。至于现在，我想仅仅点出我的观点，那就是某种意义上讲，通过合并更新和通过限制更新是同一个硬币的两面。

在这里我想通过提醒大家在第 1 章末尾写过的一些内容（在这里我要稍做注解）来结束本章。正如我在那一章所讲的，抛砖引玉的例子所展示出来的结果不言而喻的清晰[16]，然而我想说的是，把视图看作与定义它们的关系变量"相附生"的基础关系变量这种方式，是思考视图更新这个问题的一种很有效的方式。事实上，这不仅是一种很好的方式，而且我相信是逻辑上正确的。于是我的基础观点如下：首先，视图定义表达式表明了特定的约束；其次，这些约束表明了特定的补偿性操作。另外让我再重复一次，我并没有要求这些操作必须被系统管理员明确地写出来。

[16] 它们应当是很明显的，但（如我们所见）它们在特定的方面有时也会不如我们期待的那么明显，而在传统方式进行视图更新的情况下它们确实不明显。例如，在 SQL 环境下，尝试变更视图 LS 中的某些供应商所在城市的操作会成功（并且实际上会造成该供应商迁移到视图 NLS），除非视图 LS 被显式定义为"具有检查项"。

# 第 5 章

# 投影视图

更新投影？跟着指示！

——Anon：《Where Bugs Go》

现在我们来讨论一下更新投影视图时的情形，即“通过”一个投影操作来进行更新。注意：在前一章中我已经说过，限制视图和合并视图是同一枚硬币的两面。其实，对于投影视图和连接视图也是类似的关系，我们可以在本章以及接下来的 3 章中细细体味。

下面让我们来看看本章的结构是怎样的。我们的讨论会围绕一组示例来进行，准确地说是 3 个例子。其中前两个例子都在一个完全信息等价的环境中发生，第 3 个例子则会展示大家如果在信息不等价时会发生什么，即有特定的信息被隐藏的情况。对于所有 3 个例子，我都会遵循“可交换性原则”，并且开始的时候我都会假设所有的关系变量都是基础关系变量，然后我们会再来看如果其中某些变量实际是视图的话，情况会怎样。

## 5.1 示例 1：一个无损分解

首先我们确定为了简化模型，从所有供应商关系变量中去掉属性 SNAME，因此在本章范围内，“S”就变成了下面这个缩减后的关系变量。

```
S { SNO , STATUS , CITY } KEY { SNO }
```

（如你所见，我不仅去掉了属性 SNAME，而且还相应地简化了关系变量的定义。

尤其是，我已经省略掉了属性类型的具体说明，同样也是为了使示例简化易懂。）

下面，定义关系变量 ST 和 SC 为该关系变量 S 的投影。具体来说是令 ST 作为在 SNO 和 STATUS 上的投影，并令 SC 作为在 SNO 和 CITY 上的投影。

```
ST { SNO , STATUS } KEY { SNO }
SC { SNO , CITY } KEY { SNO }
```

注意观察，这两个投影组合在一起等同于平时我们所谓的对关系变量S的无损分解，在任何情况下都具有如下两种特征：（a）关系变量ST和SC的当前值等于相应关系变量S当前值的投影，（b）关系变量S的当前值等于关系变量ST和SC当前值的连接。作为这两种设计的直接结果，S自己和ST以及SC的整合是互为信息等价的[1]。注意，如定义中已经写明的，{SNO}是这 3 个关系变量的键。样本值如图 5.1 所示。

S

| SNO | STATUS | CITY |
|---|---|---|
| S1 | 20 | London |
| S2 | 10 | Paris |
| S3 | 30 | Paris |
| S4 | 20 | London |
| S5 | 30 | Athens |

ST

| SNO | STATUS |
|---|---|
| S1 | 20 |
| S2 | 10 |
| S3 | 30 |
| S4 | 20 |
| S5 | 30 |

SC

| SNO | CITY |
|---|---|
| S1 | London |
| S2 | Paris |
| S3 | Paris |
| S4 | London |
| S5 | Athens |

图 5.1　关系变量 S、ST 和 SC 的样本值

下面是 ST 和 SC 的谓词。

ST：供应商 SNO 已经签约并且拥有状态 STATUS。

SC：供应商 SNO 已经签约并且位于城市 CITY。

作者注：按照在第 2 章中的讨论，我们应当给关系变量 ST 一个更准确的谓词，那就是“供应商 SNO 已经签约并且拥有状态 STATUS‘并且位于某处’”（对于关系变量 SC 也是同样）。但是在本章中，我更愿意给大家呈现一种稍微简化的或缩短的形式。对于本章来讲这并不构成问题。

如本章开头所介绍的，我将从所有 3 个关系变量都是基础关系变量，它们之间相互附生这种情况开始介绍。当然，如果它们确实都是基础关系变量，那么我们会再次

[1] 通用版本的声明如下：设两个数据库 DB1 和 DB2，DB1 只包含一个单独的关系变量 R，而 DB2 则只包含 R 的投影。那么，为了使得 DB1 和 DB2 能够互为信息等价，最重要且必要的事情如下所示。（a）在 DB2 中的投影必须是由对 R 进行分解得来的，而该分解必须根据在 R 内生效的某些“连接依赖”（JD）来进行；（b）在任何情况下，DB2 中的每个关系变量都必须等于 R 的当前值的相应投影。大家可以参看《Database Design and Relational Theory》来了解更多背景信息。

看到一个包含有冗余的数据库，于是我们会很希望能够控制相应的冗余。接下来，一致性约束会捕捉到这些冗余（注意观察，它们都是 EQD——相等依赖条件，它们会共同作用于冗余，并保证之前提到的信息等价）。

```
CONSTRAINT ... ST = S { SNO , STATUS } ;
CONSTRAINT ... SC = S { SNO , CITY } ;
CONSTRAINT ... S  = JOIN { ST , SC } ;
```

关于第 3 条约束（也就是用前两条一起表达的那条，其中{SNO}是 S 的键），我要提醒你，从第 4 章开始 Tutorial D，至少是本书中所使用的 Tutorial D 版本，同时支持二元（中缀）和多元（前缀）的特定关系操作，尤其是包括连接。因此，例如 ST 和 SC 的连接，我们既可以用 ST JOIN SC 的方式,也可以用(上面约束)JOIN { ST , SC }的方式表达。例如我在上一章提过的，在本书中我会根据具体的写作需要分别使用这两种方式。

下面这种约束也可以用上述的前两个来表示。

```
CONSTRAINT ... IDENTICAL { S { SNO } , ST { SNO } , SC { SNO } } ;
```

这里我应该解释一下 Tutorial D 的表达式 IDENTICAL {r1,...,rn}，当且仅当所有变量 r1, ..., rn 全都相等时，它将会返回“TURE”值（可以将 IDENTICAL 看作一种“n 元等价”运算）。因此，这个约束实际上就是下列 3 个独立等式的逻辑 AND。

```
S  { SNO } = ST { SNO } AND
ST { SNO } = SC { SNO } AND
SC { SNO } = S { SNO }
```

那么会采用哪些补偿性操作呢？下面几个是显然会有的。

```
ON DELETE d FROM S : DELETE d { SNO , STATUS } FROM ST ,
                     DELETE d { SNO , CITY } FROM SC ;

ON INSERT i INTO S : INSERT i { SNO , STATUS } INTO ST ,
                     INSERT i { SNO , CITY } INTO SC ;
```

这些规则可以解决S的更新问题。那么对于ST和SC的更新呢？让我们首先来考虑一下DELETE操作。于是下面的“双DELETE”就显得非常合理了[2]。

---

[2] 这个例子是我们第一次见到用来描绘一种伪代码语法的例子，从这里之后我会经常使用这类例子。我希望它对大家来说是一个不言自明、浅显易懂的例子。不过说得具体点，例子中的表达式（S1, 20）严格讲应当写作关系选择器调用 RELATION {TUPLE {SNO ‘S1’, STATUS 20}}，而表达式（S1, London）严格来讲则应当写作关系选择器调用 RELATION {TUPLE {SNO ‘S1’, CITY ‘London’}}。

```
DELETE ( S1 , 20 ) FROM ST ,
DELETE ( S1 , London ) FROM SC ;
```

其效果就是从 ST 和 SC 中删除特定的数组，同时我们假定在合理的情况下，相应的级联删除规则都起效，这样它同时也可以从关系变量 S 中删除数组（S1, 20, London）。那么下面的这个“单 DELETE”又如何呢？具体如下。

```
DELETE ( S1 , 20 ) FROM ST ;
```

那么现在，不允许出现的情况就是：严格按照需求，只有（S1，20）被从关系变量ST中删除，而其他的一切都保持不变（因为这种结果将明确地违反我们之前列出的一些特定的约束）。其中一种可能性是单“违反黄金法则”这一项就可以直接拒绝该DELETE执行。能够按用户的需求执行确实是一件令人振奋的事情，但是，看起来还是接受DELETE并能让它进行相应的级联操作更好一些[3]。换言之，我推荐使用如下规则作为一个更合适的ST删除规则。

```
ON DELETE d FROM ST : DELETE ( S MATCHING d ) FROM S ;
```

将在投影关系变量 ST 上的删除级联到关系变量 S 将会导致之前定义对 S 的删除的规则同时进入整个过程中起效，因此最终效果将会是原始的 DELETE 也会级联到另一个投影关系变量 SC。

那么，很自然地，对于 SC 我们也需要一个类似的规则。

```
ON DELETE d FROM SC : DELETE ( S MATCHING d ) FROM S ;
```

作者注：当然，如果你不同意我所讲的观点，如上面所述通过将单一 DELETE 级联的方式能够更好地帮助它们执行成功的话，亦即，如果你选择让这类 DELETE 操作直接失败，那么我会假定你应该能通过数据库管理系统授权的子系统来实现删除效果。

那么 INSERT 操作在这些投影关系变量上的表现又如何呢？下面的“双 INSERT”看起来比较合理。

```
INSERT ( S9 , 20 ) INTO ST ,
INSERT ( S9 , London ) INTO SC ;
```

效果就是向 ST 和 SC 插入特定的数组，同样我们假定在正常情况下，相应的级联删除规则都起效（详见下面），这样它同时也可以向关系变量 S 中插入数组（S9, 20,

[3] 这里我们很有必要说明，在今天的数据库管理系统中，如果 ST 和 SC 实际上是关系变量 S 的视图，而不是分别独立的基础关系变量，那么这类级联就一定会出现。

London）。那么“单 INSERT”又如何呢？表达式如下。

```
INSERT ( S9 , 20 ) INTO ST ;
```

那么现在，不允许出现的情况就是：严格按照需求，只有（S9, 20）被插入到关系变量ST中，而其他的一切都保持不变，因为这种结果将明确地违反我们之前列出的一些特定的约束。而且，在这里没有任何级联是合理的（可以比对我们在上文中所谈到过的DELETE操作的情况），也就是说我们无法在向关系变量ST插入数组（S9, 20）的同时向关系变量S中级联插入一些（S9, 20, c）这样的数组，这正是因为给出最原始INSERT需求的用户并没有提供值c，所以我们只能自己去“完成这个数组”[4]。（由于相同的原因，对于原INSERT也没有任何向关系变量SC的级联插入是合理的。）因此初始的INSERT一定会因为违反黄金法则而失败，而唯一一个看起来合理的插入法则如下所示。

```
ON INSERT it INTO ST , INSERT ic INTO SC :
   INSERT ( it JOIN ic ) INTO S ;
```

由于这个规则，如果 it 和 ic 在 SNO 上的投影不等价，那么整个 INSERT 操作就会失败，同样也是因为违反黄金法则。由此导致的结果就是“INSERT 操作如果想成功，那么就必须使用‘双 INSERT’方式”。

为了更清楚地表达，现在让我再利用下面的扩展版重申一下前面所讲的规则，扩展版与原版是逻辑等价的，请特别注意我们介绍过的 t1 和 t2，详情如下。

```
ON INSERT it INTO ST , INSERT ic INTO SC :
   WITH ( t1 := it JOIN SC , t2 := ic JOIN ST ) :
   INSERT t1 INTO S , INSERT t2 INTO S ;
```

在这里请一定小心注意，正如在第 4 章中所说，初始的向 ST 和 SC 插入的 INSERT 操作是首先完成的，然后才是补偿性操作——两个向 S 做的 INSERT 操作，开始起效。因此，在补偿性操作中的符号“ST”和“SC”（在表达式中分别是 ic JOIN ST 以及 it JOIN SC）指的是初始 INSERT 操作被执行之后那些关系变量的值。

作者注：如你所见，扩展版规则的第 3 行，INSERT t1 INTO S, INSERT t2 INTO S，特别定义了一个双 INSERT 操作，而其中每一个 INSERT 的目标是一样的，都是关系变量 S。换句话说，这里我们首次见到了这样一个例子，由两个或更多目标是相同变量的独立赋值组成的多重赋值。简单地讲，在这种情况下，这些单独的赋值会按照语句所书写的顺序执行（请参看附录 A 获得更多详细解释）。不过对于现在手

[4] 在这个地方如果有任何人敢说“nulls”，那恐怕我不得不请他去好好“漱漱口”了。

中这个例子，我们可以通过将两个单独的 INSERT 操作整合成一个来把问题进行简化，至少概念上是可行的。于是有：

```
INSERT UNION { t1 , t2 } INTO S ;
```

不过对于这个例子，大家请一定注意，如果 t1≠t2，那么我们就会违反黄金法则。

为了方便大家阅读，现在让我来总结一下到目前为止定义过的这些规则，我会把针对 ST 和 SC 的所有的 INSERT 和 DELETE 规则合并成一个规则（为了清晰易懂，我依然还会使用已经介绍过的名字），详情如下。

```
ON DELETE dt FROM ST , DELETE dc FROM SC ,
   INSERT it INTO ST , INSERT ic INTO SC :
   WITH ( t1 := it JOIN SC , t2 := ic JOIN ST ,
          t3 := S MATCHING dt , t4 := S MATCHING dc ) :
   INSERT t1 INTO S ,
   INSERT t2 INTO S ,
   DELETE t3 FROM S ,
   DELETE t4 FROM S ;
```

同时，也让我将针对 S 的所有 INSERT 和 DELETE 规则合并为一，在这里重申一下，具体如下。

```
ON DELETE d FROM S , INSERT i INTO S :
   DELETE d { SNO , STATUS } FROM ST , DELETE d { SNO , CITY } FROM SC ,
   INSERT i { SNO , STATUS } INTO ST , INSERT i { SNO , CITY } INTO SC ;
```

作者注：由于上述两个整合规则，我们再次遇到了这种情况，那就是在某些多重赋值之中的两个或更多单独赋值所针对的目标是同一个变量。但是显而易见，在我们遇到的这几个例子中，这些单独赋值的执行顺序对于整体效果并没有任何影响。同样，请参看附录 A 获取更详细的解释。

现在，让我们考虑一下显式 UPDATE 操作。首先，我将把它留作一个例子，来展示在关系变量 S 上显式 UPDATE 都可以如大家所期待的那样起效。为了方便读者们自己检阅，大家可以考虑一下如下的例子。

```
UPDATE S WHERE SNO = ‘S1’ : { CITY := ‘Paris’ } ;

UPDATE S WHERE SNO = ‘S1’ : { SNO := ‘S9’ } ;
```

尤其请注意这个例子的第 2 行是一个“键 UPDATE”。

那么对于在 ST 和 SC 上的 UPDATE 又如何呢？让我们把下面对 ST 的 UPDATE 需

求作为一个例子来考虑。

```
UPDATE ST WHERE SNO = ‘S1’ : { STATUS := 10 } ;
```

给定我们常用的样本值，那么你将会看到如下情况发生。

1．首先，需求中的更新已经完成，也就是说，数组（S1,20）已经被从关系变量 ST 中删除，并且数组（S1,10）也已经被插入到关系变量 ST 中。

2．其次，补偿性操作已经被执行，也就是说，由于从 ST 到 S 的级联删除规则，数组（S1, 20, London）已经被从关系变量 S 中删除，而由于从 ST 到 S 的级联插入规则，数组（S1, 10, London）已经被插入到关系变量 S 中。注意：请观察在本例中，dc 和 ic 都是空的。而由此导致的结果是在对 ST 和 SC 的 INSERT 规则中，表达式 it JOIN SC 中的字符“SC”，相当于表示的是关系变量 SC 在任何更新发生之前的值，因此该表达式催生了备受需要的 S 数组（S1, 10, London）。

由此来看，这个 UPDATE 运转正常。那么下列的“键 UPDATE”又如何呢？具体如下。

```
UPDATE ST WHERE SNO = ‘S1’ : { SNO := ‘S9’ } ;
```

那么显而易见的是很不幸这个 UPDATE 操作失败了（具体来说，它由于违反了黄金法则而执行失败）。而如同上一章中供应商与设备供应的例子一样，我认为在这里别无选择，不得不明确地写出所必需的多重赋值，于是有：

```
UPDATE ST WHERE SNO = ‘S1’ : { SNO := ‘S9’ } ,
UPDATE SC WHERE SNO = ‘S1’ : { SNO := ‘S9’ } ;
```

这个“双 UPDATE”执行成功了，大家可以容易地判断。

所以，除了最后这块有点小小地恼人之外，整个事情还算在轨。（a）S 自身和对 ST 与 SC 的整合这两个设计是信息等价的，也因此有（b）对于某一个变量的每个更新，总有一个与之逻辑等价的针对另一变量的更新能够达到相同的整体效果。目前为止，一切正常。

## 5.2 示例 1 续：投影关系变量

让我们继续假设关系变量 S、ST 和 SC 都是基础关系变量，其中依然没有视图。现在我们考虑一个用户只能看到投影关系变量 ST 和 SC。那么这名用户：

1．知道相应的谓词。

ST：供应商 SNO 已经签约并且有状态 STATUS。

SC：供应商 SNO 已经签约并且位于城市 CITY。

2．了解相关的类型信息[5]，了解{SNO}是每个关系变量的键，了解下列约束。

```
CONSTRAINT ... ST { SNO } = SC { SNO } ;
```

（这是上一节中唯一一个没有涉及关系变量 S 的约束。）

3．了解如下所列的补偿性操作。

```
ON DELETE dt FROM ST , DELETE dc FROM SC ,
   INSERT it INTO ST , INSERT ic INTO SC :
   WITH ( t1 := it JOIN SC , t2 := ic JOIN ST ,
          t3 := SC MATCHING dt , t4 := ST MATCHING dc ) :
   INSERT t1 { SNO , STATUS } INTO ST ,
   INSERT t2 { SNO , CITY } INTO SC ,
   DELETE t3 FROM SC ,
   DELETE t4 FROM ST ;
```

（这个规则看起来可能有点复杂，但实际上它基本上只是对上一节中对 ST 和 SC 的 DELETE 和 INSERT 规则进行了适当的编辑，主要修改是将对 S 的引用替换为对 ST 和/或 SC 的引用。）

给出了以上所有信息后，我将把它作为一个练习，用来展示从用户角度来看，对 ST 和/或 SC 的所有更新执行时都如上一节所描述的完全一致。

## 信息隐藏

现在如果假设一个用户只能看到关系变量 ST，那又会是怎么样的呢？这个用户知道相应的谓词（详见上面所述），并且知道{SNO}是该关系变量的键，但是显然不了解任何涉及 SC 或 S 的约束。而且很有可能，该用户也不知道任何补偿性操作的存在。因此很显然，这个用户不被允许向关系变量 ST 中插入数组，也无法在该关系变量中更新供应商号码，因为这些操作都可能违反该用户所不知道的一些约束。（这里也请注意，对 ST 更新以更改供应商的状态这一操作意味着关系变量 SC 必须存在，即使该关系变量对用户隐藏。因此，如果 ST 已经被定义，而 SC 没有，特别是如果为了安全原因 ST 的定义中丝毫未涉及 SC，那么这类更新也

---

[5] 我会不厌其烦地持续说明这点，你可以把它当作一个对本书后面的所有例子都是真实、有效地内容来看待。

将被禁止执行。）

## 5.3　示例 1 续：视图

在第 4 章中，我向大家展示了，只要一直保持信息等价，即使有部分或者全部关系变量都从基础关系变量转换为视图，对整体来说也不会有本质的不同。那么在这里我要告诉大家，正如你们所期待的，这条特性在本章也一样成立，即如果与各示例有关的某些或者所有关系变量是视图的话，在本章中目前为止我所说的一切都不会因此改变。具体点说就是现在假设 S 是基础关系变量，而 ST 和 SC 都是视图，那么有：

1．在之前最后一节中由各种 CONSTRAINT 语句所指定的约束将会被自动执行，这正是因为 ST 和 SC 都是 S 的视图。

2．从 S 到 ST 和 SC 的补偿性操作也会自动发生，同样是因为 ST 和 SC 都是 S 的视图。也就是说对 S 的各种更新相应地会被自动映射到 ST 或 SC 当中。

3．从 ST 和 SC 到 S 的补偿性操作，这里有一些可以被视为视图更新规则的操作也会自动发生，同样是因为 ST 和 SC 都是 S 的视图。换言之，对 ST 或 SC 的更新“实际上”是对底层关系变量 S 的更新，并且在 S 中也会自动可见，就如在 ST 或 SC 中一样。

4．如果某个用户只能看到视图 ST 和 SC，那么他也能够实现全方位操作，就好像这些视图就是基础关系变量一样，实际上就如同上一节中它们还是基础关系变量时我所描述的一样。详细来说，该用户会知道如下几点：（a）针对每一个关系变量的所有 DELETE 操作都会级联到另一个，（b）所有 INSERT 操作和键 UPDATE 操作都必须是“双更新”模式的，（c）其他的 UPDATE 操作会正常执行，（d）所有的显式赋值都必须与前面几项合理构成的组合逻辑等价。

5．对比来说，假设某个用户只能看到视图 ST，那么他（她）所能执行的操作很明显就会很有限，就如同当所有变量都是基础关系变量时用户只能看到基础关系变量 ST 的情形一样（同样在上一节我们有过描述）。具体来说就是该用户将不会被允许向该关系变量插入数组，或者在该关系变量内更新供应商号码（并且很可能也无法改变 STATUS 值）。

## 5.4　示例 2：另一个无损分解

现在，我们来考虑示例 1 的另一个版本，或者说，是更通用的一个版本。同样，我

还是要从关系变量 S 开始谈起。只不过这一次，我会在这个关系变量中加入一个约束来实现“函数依赖”（FD）。

```
{ CITY } → { STATUS }
```

换句话说，任意两个供应商如果位于相同的城市，那么他们同时也会有相同的状态[6]。（为了让我们常用的样本值与这个新的约束一致，让我们暂时把供应商S2 的状态值从 10 改为 30。）现在，假设我们定义两个投影关系变量：SC，对于S上SNO和CITY的投影（与示例 1 中相同）；CT，对于S上CITY和STATUS的投影，详情如下。

```
SC { SNO , CITY } KEY { SNO }
CT { CITY , STATUS } KEY { CITY }
```

经过观察不难发现，如同示例 1 中的一样，这两个投影共同构成了对关系变量S的一个无损分解，在任意时刻它都具有以下特性：（a）SC和CT的值等于相应的S值的投影，（b）S的值等于表达式SC JOIN CT的值。于是，这两个定义，S和SC与CT的组合是信息等价的[7]。这些关系变量都具有单键，{SNO}是SC的单键，{CITY}是CT的单键。样本值如图 5.2 所示。

S

| SNO | STATUS | CITY |
|---|---|---|
| S1 | 20 | London |
| S2 | 30 | Paris |
| S3 | 30 | Paris |
| S4 | 20 | London |
| S5 | 30 | Athens |

SC

| SNO | CITY |
|---|---|
| S1 | London |
| S2 | Paris |
| S3 | Paris |
| S4 | London |
| S5 | Athens |

CT

| CITY | STATUS |
|---|---|
| Athens | 30 |
| London | 20 |
| Paris | 30 |

图 5.2 带{CITY} → {STATUS}的关系变量 S、SC 以及 CT 的样本值

下面是 SC 和 CT 的谓词。

SC：供应商 SNO 已经签约并且位于城市 CITY。

[6] 《Database Design and Relational Theory》一书对功能性依赖 FD 有较深入的探讨。这里我只想给大家明确，如果 K 是关系变量 R 的键，那么 FD K → X 在 R 中，对所有 R 的关系头的子集 X 起效，示例 1 其实已经很明确地引入了一些特定的 FD。

[7] 正如我在前面曾说过的，对于数据库 DB1（只包含单关系变量 R）与数据库 DB2（只包含 R 的投影）而言，当且仅当 DB2 中的投影是通过与 R 中所含的某些 JD 一致地分解得来的时候，DB1 与 DB2 是信息等价的，但是目前我们所讨论的示例 2 引入的是 FD，而不是 JD。然而，这里有一个定理——希斯定理规定，如果 A、B 和 C 的合并等于 R 的关系头，并且 FD A → B 是在 R 中起效的话，那么包含 AB (=A 与 B 的合并) 以及 AC (=A 与 C 的合并)这两个元素的 JD 也同样在 R 中起效。

CT：城市CITY拥有状态STATUS。[8]

像之前一样，让我们先来看看当 3 个关系变量都是基础关系变量时是什么情况。下列约束显然会起效。

```
CONSTRAINT ... SC = S { SNO , CITY } ;
CONSTRAINT ... CT = S { CITY , STATUS } ;
CONSTRAINT ... S = JOIN { SC , CT } ;
CONSTRAINT ... IDENTICAL { S { SNO } , SC { SNO } } ;
CONSTRAINT ... IDENTICAL { S { CITY } , SC { CITY } , CT { CITY } } ;
```

对于更新来说，S 上的 INSERT 和 DELETE 肯定会相应级联到 SC 上。

```
ON INSERT i INTO S : INSERT i { SNO , CITY } INTO SC ;
ON DELETE d FROM S : DELETE d { SNO , CITY } FROM SC ;
```

那么它们是否会级联到 CT 呢？其中，INSERT 操作可以直接级联。

```
ON INSERT i INTO S : INSERT i { CITY , STATUS } INTO CT ;
```

（我要再次提醒大家，对任何试图插入一个已经存在数组的尝试，其效果都是“noop.”）但是 DELETE 操作就会稍微复杂一点。

```
ON DELETE d FROM S :
   DELETE ( ( CT MATCHING d ) NOT MATCHING S ) FROM CT ;
```

解释说明：令 r 为由子表达式 CT MATCHING d 表示的关系。那么当给定图 5.2 所示的样本值时，有：

1. 假设我们从 S 中删除（S5，30，Athens），那么现在 S 就只包含 S1、S2、S3 和 S4。那么有（a）r 只包含一个数组（30，Athens），（b）表达式 r NOT MATCHING S 现在被调整为只有 r，因为仅有的包含（30，Athens）的子数组已经被从 S 中删除了，（c）数组（30, Athens）被从 CT 中删除了。

作者注：请注意，当我们说例如（30，Athens）是数组（S5，30，Athens）的子数组时，就意味着前者是后者的投影。更通用点的说法是如果 t 是一个供应商数组，那么 t 的属性 SNO 和 CITY 上的投影 t{SNO, CITY}就是 t 的子数组，该子数组只包含 t 的 SNO 和 CITY 元素。换句话说，我们可以，并且 Tutorial D 也支持我们方便地“重载”，给众所周知的关系投影运算定义一个用于数组的版本，就如同它们对关系的效果。请参考《SQL and Relational Theory》来获取更多信息。

[8] 因此在这个定义中，城市就变成了“它们自己本身”。

2．作为对比，假设我们从 S 中删除了（S1，20，London），那 么 S 现在只包含数组 S2、S3、S4 和 S5。于是有（a）r 只包含单独数组（20，London），（b）那么表达式 r NOT MATCHING S 也因此得到一个空关系的结果，S 仍然拥有一个数组，也就是供应商 S4，其中包含（20，London）作为一个子数组，因此（c）没有任何东西被从 CT 中删除，尤其是数组（20，London）没有被删除，并且 CT 保持原样没有变化。

并且，经我粗略观察发现，按照之前给出的补偿性操作，所有在 S 上的显式 UPDATE 操作都如预期般运行，大家如有兴趣也可以自己验证。请注意，任何想要违反 FD{CITY} → {STATUS}的尝试都会因为违反黄金法则而失败。

那么对于在 SC 和 CT 上的更新又如何呢？接下来，让我们来看一些例子，当然它们都是基于图 5.2 的样本值。首先是关系变量 SC。

1．假设我们从 SC 中删除了（S1，London），那么下列几个结果看起来就会很合理 （a）系统会从 S 中删除（S1，20，London），（b）系统不会对 CT 做任何操作（因为这里还有另一个供应商 S4，其状态是 20 并且城市为 London）。对比来看，假设我们从 SC 上删除（S5，Athens），那么如下几项应该是合理的结果。（a）系统会从 S 中删除（S5，30，Athens），（b）同时系统也会从 CT 中删除（30，Athens）。注意：当说到“合理”，我的意思当然是这些操作达到了我们所需要的效果，同时并不会违反黄金法则或者要求用户写出一个显式的多重赋值。

2．假设我们向SC中插入（S9，London）。这个更新成功，它同时有向S中插入（S9，20，London）的效果。对比来说，如果我们尝试向SC中插入（S9，Rome），那么这个更新将会失败，因为在CT中没有任何数组包含城市Rome[9]。

下面我们再来看看关系变量 CT。

1．如果我们试图向 CT 中插入（Rome，20），假设在 SC 上没有“在同一时刻”发生的更新需求（等效于在 S 上的需求，参看下面第 4 点），那么结果只能是这个操作因违反黄金法则而失败。

2．如果我们试图向 CT 中插入（London，40），那么这个尝试同样会因为违反黄金法则（实际上是违反了一个键约束）而失败。

[9] 向 SC 进行 INSERT 操作是否成功不可预测地取决于当时数据库系统的状况，并且可能因此被拒绝会让你觉得有些失控。但是当更新失败的时候我们至少知道原因是它们违反了一些用户明确知道的约束条件。实际上，上述更新失败与我们传统意义上的因为违反关键约束而发生的更新失败并没有本质上的不同，后者在发生时其实也是取决于数据库当时的情况。

3．如果我们从CT中删除（London，20），那么应该会有如下结果。（a）从S中删除（S1，20，London）和（S4，20，London），（b）从SC中删除（S1，London）和（S4，London）[10]。

4．如果我们“在同一时刻”向 SC 插入（S9，Rome），并向 CT 插入（Rome，20）。那么这个更新将会成功，它同时具备向 S 中插入（S9，20，Rome）的效果。

根据这些例子，我们可以总结出以下几个规则。

```
ON DELETE dc FROM SC , DELETE dt FROM CT :
   DELETE ( S MATCHING dc ) FROM S ,
   DELETE ( S MATCHING dt ) FROM S ;

ON INSERT ic INTO SC , INSERT it INTO CT :
   INSERT ( ic JOIN CT ) INTO S ;
```

*后面一条规则可以被无损简化为如下的形式（为什么无损？）。*

```
ON INSERT i INTO SC :
   IINSERT ( i JOIN CT ) INTO S ;
```

有了这些规则，前面提到的所有的 INSERT 和 DELETE 操作的例子都可以正常执行，并获得大家所期待的效果。同样，根据我的观察，显式 UPDATE 操作也会如预期地正常运行（关于这点的验证就留给大家当作练习了）。对于示例 1 来说，状况就是：（a）对于两个不同的设计 S 本身，及由 SC 和 CT 整合而来的形式来说，它们是信息等价的，也因此会（b）对其中任何一个做出更新，那么理论上在另一个上面会有一个等价的更新被执行，并达到总体相同的效果。

作者注：现在，我终于可以解决前几章遗留下来的一点未尽之事了，就是那些对于视图LS（“London suppliers”）还未解决的问题。在前几章中，我曾说过我们可以从该关系变量中去掉CITY属性。实际上，我们可以把该关系变量分解为它的几个投影：（a）SNT，有属性SNO、SNAME，以及STATUS；（b）C，有属性CITY。当然，后面这个投影只拥有一个数组，包含单独的一个值London。）正如 在《Database Design and RelationalTheory》中所介绍的，FD {} → {CITY}在关系变量LS中起效[11]，因此，早前在本章的一个注脚中提到过的希斯定理，将会保证该分解为无损

[10] 如果你对于在 CT 上进行删除操作会级联到 S 这个主意感到有那么一点不舒服的话，那么请记住现在城市属性都已经是“它们自己本身”了。同理，对于在 S 上进行删除会级联到 SP 上也是一样，本质上这些情况是相似的。
[11] 顺便说一句，那个 LS 根本都不算是第二范式（2NF）。

分解。那么关系变量SNT就可以根据以下规则进行更新。

```
ON INSERT i INTO SNT : INSERT ( i JOIN C ) INTO LS ;

ON DELETE d FROM SNT : DELETE ( d MATCHING C ) FROM LS ;
```

当然，前提是关系变量 LS 是可更新的。注意：我将把对于关系变量 C 的更新的相关观察和思考作为练习留给你们。

下面，为了文章的完整性，我会像在示例 1 中那样写出我对示例 2 的一些观点和思考角度，如下：（a）单独思考关系变量 SC 和 CT，不考虑关系变量 S 的情况；（b）考虑如果用户只能看到这两个关系变量中的一个，即如果部分信息被隐藏了，那么将会发生什么情况；（c）思考如果 SC 和 CT 其实都是视图，那么会出现什么情况。不过我想，平心而论，对于这几个问题并不会有什么令人意外的结果，如果真的有意外的结果，那才叫意外。实际上，请大家将注意力集中到一个更重要的点上，也就是，你们可能已经注意到了，对于示例 2 我所给出的更新规则其实都可以对应到示例 1 相应的元素中去，或者，让我们说得更准确一点，那就是示例 1 的规则都是示例 2 规则的特殊情况。除此之外，只有在示例 1 所描述的情况是示例 2 所描述的情况的特例时，这种规则关系才会出现。（回想一下，示例 1 中也引入了对 S 的无损分解，只是它被放进了不同的投影中。）

### 5.4.1 投影关系变量

我们继续假设关系变量 S、SC 以及 CT 都是基础关系变量，还是没有视图。那么现在我们来看看当一个用户只能看到投影关系变量 SC 和 CT 时的情形（同时我又观察了一下，由 SC 和 CT 这两个关系变量整合的方式与只包含关系变量 S 的方式是信息等价的）。对于该用户会有以下情况出现。

1．知道各关系变量相应的谓词。

SC：供应商 SNO 已经签约并且位于城市 CITY。

CT：城市 CITY 拥有状态 STATUS。

2．了解{SNO}是 SC 的键，{CITY}是 CT 的键，并且也了解如下的约束。

```
CONSTRAINT ... IDENTICAL { SC { CITY } , CT { CITY } } ;
```

3．了解如下这些补偿性操作。

```
ON DELETE dc FROM SC , DELETE dt FROM CT :
   DELETE ( SC MATCHING dt ) FROM SC ,
   DELETE ( CT MATCHING dc ) NOT MATCHING SC ) FROM CT ;
```

给出了以上信息之后，我将会把以下工作作为练习留给大家：验证从这个用户的角度来看，所有对于 SC 和/或 CT 的更新都会如其预期的一样被正常执行。

### 5.4.2 信息隐藏

那么如果一个用户只能看到关系变量 SC 又会是什么情形呢？答案是该用户知道谓词是怎样的（看上面所列的），并且还知道{SNO}是该关系变量的键，不过当然不会知道任何与 CT 或 S 有关的约束。再多说一点，就是该用户也不知道任何补偿性操作。而这种情形就会导致：如果新数组或更新的数组中的 CITY 值不是已经存在于该关系变量中的，那么该用户就不被允许执行任何 INSERT 或者 UPDATE 操作。至于只能看到关系变量 CT 的用户，则不被允许对 CITY 值做出任何 INSERT 或 UPDATE 操作。当然，有此限制的原因又是该用户只能看到部分而非全局，在这种情况下信息等价也无法成立。

### 5.4.3 视图

最后，如果本章涉及的部分或者全部关系变量都变为视图，那么目前为止的所有结论仍然适用，毫无例外。我们来具体说说，假设现在 S 是基础关系变量，而 SC 和 CT 是视图，那么有：

1. 在本节早些时候用 CONSTRAINT 声明定义的各种约束都会自动强制执行，而原因正是 SC 和 CT 都是 S 的视图。

2. 所有从 S 到 SC 和 CT 的补偿性操作也都会自动发生，这同样是因为 SC 和 CT 都是 S 的视图，也就是说，所有对 S 的更新都会自动在 SC 或 CT 或两者中相应地体现出来。

3. 所有从 SC 和 CT 到 S 的补偿性操作，也就是那些通常可以理解为视图更新规则的部分，同样也会自动运行，而原因也同样是 SC 和 CT 都是 S 的视图。也就是说，对于 SC 和 CT 的更新是“真正的”对底层关系变量 S 的更新，而它们在 S 中也是自动可见的，就如同在 SC 或 CT 中一样。

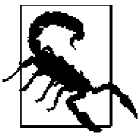
4. 一个只能看到视图 SC 和 CT 的用户可以像它们是基础关系变量一样执行所有行动，实际上，与上面它们就是基础关系变量的例子中所描述的完全一样。

5. 对比而言，假设一个用户只能看到视图 SC，那么他或她显然会被限制执行某些操作，就如同当 3 个变量都是基础关系变量时，该用户只能看到基础关系变量 SC 一样。

## 5.5 示例 3：一个有损分解

下面我想最后再换一个例子，这也是本章中我想讨论的最后一个例子。前面几个例子我都是从关系变量 S 开始的，而这次不同，我将假设（与示例 2 相比）函数依赖 FD{CITY}→{STATUS}并不存在。那么现在让我们来定义两个投影关系变量：ST，在属性 SNO 和 STATUS 上的投影，以及 CT，在属性 STATUS 和 CITY 上的投影。

```
ST { SNO , STATUS } KEY { SNO }
TC { STATUS , CITY } KEY { STATU S , CITY }
```



关系变量 ST 与其在示例 1 中的情况完全一样。而关系变量 TC，相比来说，虽然与示例 2 中的关系变量 CT 具有相同的关系头，但是却具有不同的语义，这一点可以通过它们拥有不同的键来体现。也正因为这个原因，我将给它起一个不同的名字，以期能够降低大家混淆的风险。样本值如图 5.3 所示。注意观察由于 FD {CITY} → {STATUS}不再存在，我已经将供应商 S2 的状态值改回了 10。

S

| SNO | STATUS | CITY |
|---|---|---|
| S1 | 20 | London |
| S2 | 10 | Paris |
| S3 | 30 | Paris |
| S4 | 20 | London |
| S5 | 30 | Athens |

ST

| SNO | STATUS |
|---|---|
| S1 | 20 |
| S2 | 10 |
| S3 | 30 |
| S4 | 20 |
| S5 | 30 |

TC

| STATUS | CITY |
|---|---|
| 20 | London |
| 10 | Paris |
| 30 | Paris |
| 30 | Athens |

图 5.3　关系变量 S、ST 以及 TC 的样本值

如前面定义所呈现的，ST的单键是{SNO}，而TC的单键是{STATUS,CITY}（整个关系头）。请注意，与示例 1 和 2 相比，关系变量ST和TC一起构成了一个“有损的”，而不是无损的对关系变量S的分解。换句话说也就是S并不等价于ST和TC的连接[12]，而这两种设计，S自己，以及ST与TC的整合，则并不是信息等价的，说得更具体点就是信息等价所依仗的那些供应商所在的城市在后面一种设计中丢失了。

那么接下来，我们还是如同前几个示例一样先假设所有的 3 个关系变量都是基础关系变量。于是就有对于 ST 和 TC 的谓词如下。

ST：供应商 SNO 已经签约并且拥有状态 STATUS。

[12] 这是因为 ST 和 TC 并没有从 S 中获得任何相应的 JD（连接依赖）。相关内容还是请大家参看《Database Design and Relational Theory》来获得更多的说明。

TC：某些供应商已经签约并且拥有状态 STATUS，同时位于城市 CITY。

作者注：在这里我想强调，本次对关系变量 S 的分解中含有关系变量 TC，是一次有损分解，我打破在本章中的惯例而给了该关系变量一个量化的形式（“某些供应商……”）。

下列几个约束显然是起效的。

```
CONSTRAINT ... ST = S { SNO , STATUS } ;
CONSTRAINT ... TC = S { STATUS , CITY } ;
CONSTRAINT ... S { SNO } = ST { SNO } ;
CONSTRAINT ... ST { STATUS } = TC { STATUS } ;
```

接下来，一个分析告诉我们对 S 的更新规则，这与我们曾经在示例 2 中做过的分析本质上是很相似的，至于细节就留给大家做练习了。具体规则如下。

```
ON INSERT i INTO S :
   INSERT i { SNO , STATUS } INTO ST ,
   INSERT i { STATUS , CITY } INTO TC ;

ON DELETE d FROM S :
   DELETE d { SNO , STATUS } FROM ST ,
   DELETE ( ( TC MATCHING d ) NOT MATCHING S ) FROM TC ;
```

那么对于在 ST 和 TC 上的更新又如何呢？我们还是先来考虑下面几个例子。

1．假设我们从 ST 中删除（S1，20），那么下面几种结果看起来是成立的。（a）会从 S 中删除（S1，20，London），（b）对 TC 没有任何操作（因为这里有另一个供应商 S4，拥有状态 20，并位于城市 London）。对比来说，假设我们从 ST 中删除（S2，10），那么就会有（a）会从 S 中删除（S2，10，Paris），（b）会从 TC 中删除（10，Paris）。

2．假设我们试图向 ST 中插入（S9，20）。带入我们通常使用的样本值，这个更新操作成功，它会有向 S 中插入（S9，20，London）的效果。对比来说，假设我们试图向 ST 插入（S9，30），那么该操作会因为违反一个键约束而失败，因为它试图为供应商 S9 向该关系变量中同时插入两个不同的数组。类似的，如果我们试图向 ST 中插入（S9，40），那么该更新同样会失败，不过这次是因为在 TC 中没有任何一个数组的状态值是 40。

3．假设我们试图向 TC 中插入某些新数组，或者从 TC 中删除某些已经存在的数组。如果在 ST 上没有“在同一时刻”发生的更新需求（等效于在 S 上的需求，参看下面第 4 点），那么结果只能是因为违反了黄金法则，该更新失败。

4．假设我们“在同一时刻”向 ST 插入（S9，40），并向 TC 插入（40，Rome），那么这个更新将会成功，它同时具备向 S 中插入（S9，40，Rome）的效果。类似的，假设我们“在同一时刻”从 ST 中删除（S2，10），而从 TC 中删除（10，Paris），那么这个更新也会成功，同时它具有从 S 中删除（S2，10，Paris）的效果，只不过在这个例子中，看起来在 TC 上的 DELETE 操作不论我们是否提出需求它都会发生（参看上面第 1 点）。

通过思考上述内容，我们得出了如下的规则。

```
ON DELETE d FROM ST :
   DELETE ( S MATCHING d ) FROM S ;

ON INSERT i INTO ST :
   INSERT ( i JOIN TC ) INTO S ;
```

有了这些规则，前面提到的所有的 INSERT 和 DELETE 操作的例子都可以正常执行，并获得大家所期待的效果。然而，显式 UPDATE 操作却成为了一个问题。实际上，我们发现大部分这类操作，不论是在 ST，或是 TC 上，都会出现要么失败，要么效果不理想的情况。举例来说，你可以想想如果我们试图更新 ST，将供应商 S2 的状态改为（a）40，（b）20，（c）30，或者将供应商 S1 的号码改为 S9，那么将会发生什么。（当然，所有这些类似的操作在关系变量 S 上肯定都可以完成。）出现这种情况的原因当然就是这两种不同的设计，S 自己，或者是 ST 同 TC 的整合，其二者并不是信息等价的。更具体点来说，让我们再来讲一次，那就是位于给定城市中相应给定的供应商可以在第一种设计中体现，而一般情况下在后一种设计中却不能体现。因此，这种显式的 UPDATE 操作还不如就根本不提。

接下来，我要通过以下几方面来总结我对这个示例的观察与分析：（a）将投影关系变量独立于关系变量 S 之外进行思考，（b）考虑如果我们只能看到这些投影关系变量其中之一的话会发生什么，（c）考虑一下如果这些关系变量是视图的话会发生什么。不过，我相信你依然无法发现任何“惊喜”。而唯一你可能发现的一点“惊喜”，也不过是我所给出的对本示例的更新规则，与在示例 1 和示例 2 中所给出的规则即使不是完全一样的话，也会非常的相似。（在这里再重复一遍，它们确实很相似，但是泛泛地讲，比起在示例 1 和 2 中，更新在示例 3 中更容易失败。）我个人认为，在本章中所讨论的这些示例已经足够为一般的投影更新定义规则了。

## 5.5.1 投影关系变量

让我们假设一个用户只能看到投影关系变量 ST 和 TC，那么这个用户：

1．知道各变量相应的谓词。

ST：供应商 SNO 已经签约并且拥有状态 STATUS。

TC：某些供应商已经签约并且拥有状态 STATUS，同时位于城市 CITY。

2．了解{SNO}是关系变量 ST 的键，而 TC 是“全键”，并且也了解下面的约束。

```
CONSTRAINT ... ST { STATUS } = TC { STATUS } ;
```

3．了解对于 TC 的更新总是失败，原因是缺乏一个相应的“在同一时刻”发生的对关系变量 ST 的更新，也了解在 ST 上的 DELETE 操作受制于下列规则。

```
ON DELETE d FROM ST :
   DELETE ( ( TC MATCHING d ) NOT MATCHING ST ) FROM TC ;
```

请注意，这里并没有对于 ST 的 INSERT 规则。实际上，这类操作将会失败，除非下面两点同时满足：（a）在每一个新数组中 SNO 的值当前都不出现在 ST 中；（b）在每一个新数组中 STATUS 的值当前都出现在 TC 中，但只有一次。对于显式 UPDATE 操作，请参考上一节中从第二段到最后一段的说明。

## 5.5.2　信息隐藏

显然，示例 3 已经引入了信息隐藏，因为在这里我们并不具备信息等价。不过我们还可以隐藏更多信息，只要使用户只能看到两个投影关系变量中的一个，不论是 ST 或 TC 中的哪一个。那么在这种情况下用户能够执行哪些操作呢？如果是 ST，那么用户可以执行 DELETE 操作，其他都不行；如果是 TC，那么任何更新都不能执行。

## 5.5.3　视图

最后，如果本章涉及到的部分或者全部关系变量都变为视图，那么目前为止的所有结论仍然适用，毫无例外。我们来具体说说，假设现在 S 是基础关系变量，而 ST 和 TC 是视图，那么有：

1．在本节早些时候用 CONSTRAINT 声明定义的各种约束都会自动强制执行，而原因正是 ST 和 TC 都是 S 的视图。

2．所有从 S 到 ST 和 TC 的补偿性操作也都会自动发生，这同样是因为 ST 和 TC 都是 S 的视图，也就是说，所有对 S 的更新都会自动在 ST 或 TC 或两者中相应地体现出来。

3．所有从ST和TC到S的补偿性操作，也就是那些通常可以理解为视图更新规则的

部分，同样也会自动运行，而原因也同样是ST和TC都是S的视图[13]。也就是 说，对于ST和/或TC的更新是“真正的”对底层关系变量S的更新，而它们在S中也是自动可见的，就如同在ST或TC中一样。

4．一个只能看到视图 ST 和 TC 的用户可以像它们是基础关系变量一样执行所有行动，准确地讲，恰恰与上面它们都是基础关系变量的例子中所描述的完全一样。

在这里有一点你可能觉得略有些奇怪，那就是关于显式 UPDATE 操作。我之前说过在关系变量 ST 中试图改变供应商状态的尝试通常来讲要么会失败，要么效果不理想的情况，也因此这类需求其实压根就不应该提出。然而，按照传统的方式来进行视图更新的话，这种操作却根本不是问题，也就是说这类操作能够按照预期正常执行。但是，这是因为传统方式默认具备信息等价的条件。也就是说，对于这个例子来说，传统方式相当于认为我们所面对的是由 ST 和 SC 组成的形式（也就是本章示例 1 的情况），而不是我们所讨论的由 ST 和 TC 组成的形式。而在示例 1 中，大家应该还记得，试图在关系变量 ST 中改变供应商状态的操作可以正常执行。

5．对比而言，假设一个用户只能看到视图 ST 或 TC，那么他或她显然会被限制执行某些操作，就与本节前面所讲的完全一样。

# 5.6 小结

这里还有最后两点结论我要总结。首先，针对投影视图更新的话题我并没有说完，我将在第 7 章“重温投影视图”一节中再来概括一下这个主题。第二，我要澄清一些早该澄清的事情……在第 1 章，我曾经说过所有的视图都是可更新的这个观点，而在这一章我则说过针对特定视图的特定更新无论如何也是无法奏效的。作为一个极端的例子，视图 TC 被定义为对关系变量 S 中 STATUS 和 CITY 元素的投影，在这个例子中函数依赖 FD {CITY} → {STATUS}并不存在，该视图完全不能被更新。那么我是一嘴说两边，黑白全占上了吗？这到底是怎么回事呢？

另外，我在第 1 章还说过在之后会对“这个吸引眼球的想法”（就是视图都是可更新的这点）做详细说明。那么重点在这里：并不是说这个特定的视图从本质上是不可更新的。而是说，如果这些更新被系统接受，那么它们会造成一些一致性约束被

[13] 实际上，这里并没有任何针对关系变量 TC 的补偿性操作，当然这点并不影响我们所说的概念成立。

违反，并且没有补偿性操作在这里能够防止这些违反约束的情况发生，所以在这些特定视图上的这些特定更新才会失败。请注意，这个情况与对象是基础关系变量时发生的情况是一样的，即针对特定基础关系变量的特定更新也会失败，并不是因为这些关系变量从本质上讲是不可更新的，而是因为这些更新会导致一致性约束被违反。例如向设备供应关系变量 SP 中插入一个并不存在的供应商的数组，就会出现这样的状况。

# 第 6 章

# 连接视图 I：一对一连接

这段话为谁编，正是为了连接视图——更新通过 V 来做，更新 A 同时更新 B。

——Anon：《Where Bugs Go》

我想通过回顾我在第 3 章末尾所讲过的话来开启本章。

很不幸，视图更新这个话题确实很容易让人迷惑，即使在它不存在争议的时候也一样。一些难点会横在我们的研究之路当中，无可避免地成为研究的障碍，并且很容易让人在争议和讨论中渐渐失去方向。尤其是我们很容易忘记，特别是在范例当中，哪些关系变量是基础关系变量，哪些是视图。保持头脑清晰很重要！

我在这里回顾这段文字是因为我觉得只有这段话才能告诫大家，这些问题在研究连接视图的过程中尤其容易出现。

通过这段话，我想在我们这个主题的开头给大家一些感官上的概念，是会对大家有帮助的。我们的目标，毫无疑问是一系列通过连接进行更新的规则。而我所说的我们在结束时解决问题所要用到的正是一系列规则——一套规整的规则，它对所有的连接都有效[1]。尤其要说的是无论我们面对的连接是一对一的、一对多的，还是多对多的，对这套规则来说都不会有任何不同。实际上，我已经迫不及待地想要给大家展示一下我所说的这套规则到底长什么样

[1] 有些人可能会对“有效”这种说法打个问号。我所说的意思是：在所有情况下，这套规则至少是被良好定义的，并且结果是可预测的。至于这些结果是否都是可接受的，那就是另一个问题了！后面我还会做进一步讨论。

了。假设我们尝试更新一个被定义为A JOIN B的视图V。那么这套规则可以表达为如下形式。

```
ON INSERT INTO V : INSERT A (sub)tuples if they don't already exist,
                   INSERT B (sub)tuples if they don't already exist

ON DELETE FROM V : DELETE A (sub)tuples if they don't exist elsewhere,
                   DELETE B (sub)tuples if they don't exist elsewhere
```

当然，这些规则现在看起来还非常模糊不清，不过接下来在本章以及后面两章中我们将为大家呈现一幅宏篇大论，在我们努力进行细节讨论时，我相信这个粗糙的准公式能够给我们提供不少帮助。

## 6.1 示例1：信息等价

因为连接是如此的重要，又因为我们有如此多关于它的内容要讨论，所以我决定将这部分内容分为3章来讲述。在6.1节，我想把我的注意力集中在最简单的一种情况下，也就是连接是一对一的情况。按照惯例，我将从一些简单的例子开始讲起。

我的第一个例子是将上一章中关于投影的第1个例子反转过来。（正如上一章所说过的，通常来讲连接视图和投影视图之间是有紧密联系的。）让我们来具体看看，假设我们给定基础关系变量ST和SC。大致情形如下。

```
ST { SNO , STATUS } KEY { SNO }
SC { SNO , CITY } KEY { SNO }
```

（在第5章中，我为了简化例子而省略了属性SNAME。）那么下面假设我们定义这两个关系变量的连接ST JOIN SC为视图S。

```
S { SNO , STATUS , CITY } KEY { SNO }
```

接下来，让我们假定这个连接是严格的一对一的形式，在这种情况下ST中的每一个数组都会与SC中的一个数组连接，反之亦然。换句话说，在这里我们有信息等价，即由ST和SC一起组成的设计，与只包含视图S的设计显然是信息等价的。样本值如图6.1所示（当然，图6.1与图5.1是相同的，只不过现在ST和SC是基础关系变量，而S则是一个视图）。

下面是这个连接的谓词，当然它基本上就是平时关系变量S的谓词，但是为了简化例子我还是将SNAME去掉了，详情如下。

S：供应商SNO已经签约，拥有状态STATUS，并且位于城市CITY。

S

| SNO | STATUS | CITY |
|---|---|---|
| S1 | 20 | London |
| S2 | 10 | Paris |
| S3 | 30 | Paris |
| S4 | 20 | London |
| S5 | 30 | Athens |

ST

| SNO | STATUS |
|---|---|
| S1 | 20 |
| S2 | 10 |
| S3 | 30 |
| S4 | 20 |
| S5 | 30 |

SC

| SNO | CITY |
|---|---|
| S1 | London |
| S2 | Paris |
| S3 | Paris |
| S4 | London |
| S5 | Athens |

图 6.1　关系变量 S、ST 以及 SC 的样本值

按照惯例来讲，我们首先要做的就是考虑如果 3 个关系变量全是基础关系变量时会发生什么。但是我们已经在第 5 章讨论示例 1 的时候做过这件事了！因此，我在这里仅会重复一下关于这个讨论的一些重点内容。首先，我们已经说过，{SNO}是每一个关系变量的单键。那么下面的约束也是起效的（正如在第 5 章中所说的，它们都是相等依赖条件（EQD），在一起它们可以保证两种不同的设计是信息等价的）。

```
CONSTRAINT ... S  = JOIN { ST , SC } ;
CONSTRAINT ... ST = S { SNO , STATUS } ;
CONSTRAINT ... SC = S { SNO , CITY } ;
CONSTRAINT ... IDENTICAL { S { SNO } , ST { SNO } , SC { SNO } } ;
```

实际上，如果你翻回去查看的话会发现，在上一章我是用不同的顺序排列这些约束的。在这里我调整了它们的排序是为了向大家展示以下几点：首先，S 确实是等价于另外两个关系变量的连接；其次，为了保证我们需要的信息等价，ST 和 SC 确实等价于相应的 S 的投影；最后，任何一个在 S 中出现的供应商号码都会同时在 ST 和 SC 中出现，反之亦然。注意：实际上最终的那个约束是其他 3 个约束放在一起之后的逻辑结果（想想看，到底是哪 3 个）。

补偿性操作也是与之前一样，不过同样的在这里我也调整了顺序。

```
ON DELETE dt FROM ST , DELETE dc FROM SC ,
   INSERT it INTO ST , INSERT ic INTO SC :
   WITH ( t1 := it JOIN SC , t2 := ic JOIN ST ,
          t3 := S MATCHING dt , t4 := S MATCHING dc ) :
   INSERT t1 INTO S ,
   INSERT t2 INTO S ,
   DELETE t3 FROM S ,
   DELETE t4 FROM S ;

ON DELETE d FROM S , INSERT i INTO S :
   DELETE d { SNO , STATUS } FROM ST ,
   DELETE d { SNO , CITY } FROM SC ,
   INSERT i { SNO , STATUS } INTO ST ,
   INSERT i { SNO , CITY } INTO SC ;
```

这里让我来提醒大家一下，在 ST 和 SC 上，INSERT 和“键 UPDATE”操作都必须是“双更新”，如若不然，它们就会因违反黄金法则而失败。

接下来，我们就来看看如果 S 实际上是 ST 和 SC 的连接视图的话会出现什么情况。那么会有：

1．实际上，S 是 ST 和 SC 的视图这一点并不足以保证上述各种各样的约束可以被自动强制执行（第 1 条除外）。而补偿性操作也是不足以保证约束可以自动强制执行的。举个例子，不论 S 是 ST 和 SC 的连接这一点，还是补偿性操作，都不足以防止关于供应商 S9 的数组同时向 ST 和 SC 进行插入操作。因此，相应的约束需要进行显式声明，当然这个动作需要数据库管理系统来强制执行。

2．不过，从 ST 和 SC 到 S 的补偿性操作则会自动发生，这正是因为 S 是一个视图。也就是说，在 ST 和/或 SC 上发生的更新会自动反映到 S 中相应的部分。

3．从 S 到 ST 和 SC 的补偿性操作，也就是可以被视为视图更新规则的那些操作，也会自动发生，这也是因为 S 是一个视图。也就是说，对于 S 的更新是“真正的”对底层关系变量 ST 和/或 SC 的更新，并且也会在 S 中自动可见，正如其在 ST 或 SC 中一样。

同样，如果某个用户只能看到连接视图 S，那么他也能够实现全方位操作，就好像这个视图就是基础关系变量一样。这样的用户将会知道{SNO}是一个键（但并不知道其他的约束，或者任何的补偿性操作）同时也会知道相应的谓词。

S：供应商 SNO 已经签约，拥有状态 STATUS，并且位于城市 CITY。

另外，如果 S 是一个基础关系变量的话，那么所有对视图 S 的 INSERT、DELETE 以及显式 UPDATE 的操作都可以完全按照预期来完成。

## 6.2 示例 2：信息隐藏

目前来看，在有信息等价这个前提的情况下，我们想要更新一个一对一的连接视图本质上来说是非常简单的。但是如果我们遇到的是有信息隐藏存在的情况时又会怎样呢？让我们举一个具体的例子来看一看，假设一个供应商只能在基础关系变量 ST中存在，而不能存在于基础关系变量SC中或用其他方式表示，换句话说就是区别于示例 1 中每一个供应商都必须同时具有状态和城市两个属性的情况，现在我们假设每一个供应商都至少要具有这两个属性当中的一个 [2]。如果这样的话，那么ST

[2] “至少具有一个”，因为允许一个供应商什么属性都不具备，这样显然没什么道理——这样的供应商既不会在 ST 中出现，也不会在 SC 中出现，因此也不会在 S（= ST JOIN SC）中出现。

和SC的连接S就会丢失信息，这是因为，在这个情况下，系统不再能够保证ST和SC的相应属性一定等价于该连接的投影。实际上，我们在上一节中所提到的那些约束中，只有那些普通的键约束能够保证继续有效（{SNO}是S、ST和SC的键），以及下面这一条[3]。

```
CONSTRAINT ... S = JOIN { ST , SC } ;
```

满足以上条件的样本值可以通过修改图 6.1 的内容获得，从关系变量 S 和 ST 中移除供应商 S5，而在 SC 中保留，如图 6.2 所示。当然，S 的谓词依然与示例 1 中相同。

S

| SNO | STATUS | CITY |
|---|---|---|
| S1 | 20 | London |
| S2 | 10 | Paris |
| S3 | 30 | Paris |
| S4 | 20 | London |

ST

| SNO | STATUS |
|---|---|
| S1 | 20 |
| S2 | 10 |
| S3 | 30 |
| S4 | 20 |

SC

| SNO | CITY |
|---|---|
| S1 | London |
| S2 | Paris |
| S3 | Paris |
| S4 | London |
| S5 | Athens |

图 6.2　图 6.1 的修改版

关于术语的注释：从本章的题目可以推断，或者至少它强烈暗示了 ST JOIN SC 仍然是一个一对一的连接，但很可惜它并不是。实际情况是术语“一对一”经常被不严谨地用于描述下面两类情况：（a）比如上一节中的示例 1，这类应当被称为严格符合类型；（b）就像现在我们在讨论的情况，在这种情况下可能出现其中一方包含一种元素是另一方所没有的。下面是一段轻微编辑过的内容，摘自《The Relational Database Dictionary, Extended Edition》（Apress，2008）。

一个一对一的对应，严格来讲是将两个集合 S1 和 S2 进行配对的规则，其中每个 S1 中的元素应当与 S2 中的元素一一对应，而 S2 中的每一个元素也应当与 S1 中的元素一一对应。但是，这个术语却经常被不严谨地用于下面这几种配对情况：（a）每个 S1 中的元素对应 S2 中至多一个元素，而 S2 中的每一个元素都与 S1 中的元素一一对应，或者（b）S1 中的每一个元素都与 S2 中的元素一一对应，而 S2 中的每一个元素则与 S1 中的至多 1 个元素对应，或者（c）S1 中的每一个元素与 S2 中的至多一个元素对应，S2 中的每一个元素与 S1 中的至多一个元素对应。除非想要表达的意思非常明确，否则我们最好避免使用这个术语。

虽然文中最后一句这么说（建议避免使用），但我还是会在接下来的文章中频繁地使用这个术语，因为我认为根据上下文来看，我所要表达的意思始终都很明确。注

[3] 特定的包含依赖条件（IND）也会起效，但我不会去展开讨论它们的细节。

释结束。

让我们回到示例 2。让我来强调一下现在的状况，正是因为连接视图 S 丢失信息，所以一个只包含该视图的数据库无法信息等价于包含基础关系变量 ST 和 SC 的数据库。这里举一个后者中元素在前者中没有对应元素的查询例子，那就是“取得有城市属性而没有状态属性的供应商的供应商号码”。更准确地说，任意可以通过 S 单独表达的信息确定都可以通过 ST 和 SC 的整合来表达，但是反过来却不行。这种情况的结果就是显然会存在对 ST 和/或 SC 所做的更新，在 S 中没有对应的操作。下面这种操作就是个例子：执行“向 SC 中插入以 S9 为供应商号码的某个供应商的数组，城市是 London”的操作，而并不在同一时刻向 ST 插入相同供应商 S9 的数组。

那么哪些补偿性操作是适用的呢？既然在关系变量 ST 和 SC 之间不再有严格的一对一关系，那么现在就有可能像前面刚刚说的那种情况，也就是向 ST 或 SC 其中一个关系变量进行插入或删除操作，而并没有在“同一时刻”针对另一个关系变量发起相同操作。因此，在 S 被定义为必须在所有时间都要等价于 ST 和 SC 的连接的前提下，我们显然可以得到下列几种。

```
ON INSERT it INTO ST , INSERT ic INTO SC :
   INSERT ( it JOIN SC ) INTO S ,
   INSERT ( ic JOIN ST ) INTO S ;

ON DELETE dt FROM ST , DELETE dc FROM SC :
   DELETE ( S MATCHING dt ) FROM S ,
   DELETE ( S MATCHING dc ) FROM S ;
```

为了更容易阅读，我已经分类向大家介绍了这些规则，不过它们当然可以整合成一个。

```
ON DELETE dt FROM ST , DELETE dc FROM SC ,
   INSERT it INTO ST , INSERT ic INTO SC :
   DELETE ( S MATCHING dt ) FROM S ,
   DELETE ( S MATCHING dc ) FROM S ,
   INSERT ( it JOIN SC ) INTO S ,
   INSERT ( ic JOIN ST ) INTO S ;
```

如果你将这些规则同上一节中示例 1 的相应规则进行对比（那恰恰是个严格的一对一的例子）你会发现它们本质上其实是一回事（除了我在前一节中为了方便阅读使用了一些已经介绍过的变量名字之外）。下面为了方便大家对比，此处将之前的规则罗列如下。

```
ON DELETE dt FROM ST , DELETE dc FROM SC ,
   INSERT it INTO ST , INSERT ic INTO SC :
   WITH ( t1 := it JOIN SC , t2 := ic JOIN ST ,
```

```
        t3 := S MATCHING dt , t4 := S MATCHING dc ) :
    INSERT t1 INTO S ,
    INSERT t2 INTO S ,
    DELETE t3 FROM S ,
    DELETE t4 FROM S ;
```

目前关于在 ST 和/或 SC 上的更新就这么多。不过对于连接视图又如何呢？也就是，如果我们对 S 进行 INSERT 或 DELETE 操作，而不是对 ST 和/或 SC 的话，会是怎样的情况呢？实际上 INSERT 操作可以很直接地进行。

```
  ON INSERT i INTO S :
    INSERT i { SNO , STATUS } INTO ST ,
    INSERT i { SNO , CITY } INTO SC ;
```

注意，当我们向S插入数组t时，一定会同时级联到ST和SC。确实向ST和SC的级联INSERT操作之一可能是“空操作”，但它们不可能都是[4]。另外还请注意以下两种情况：（a）如果S是视图，也就是我们目前讨论的情况，而假设我们仅仅级联到ST，并且在SC中并不存在一个与之一致的数组，那么数组t则永远不会出现在S中，并且我们会违反“赋值原则”；（b）反过来讲，如果 3 个关系变量都是基础关系变量，并且我们也只是级联到ST，同时SC中不存在与之一致的数组，那么S将不再等价于ST JOIN SC，而这次我们将会违反“黄金法则”。上述两种不同情况所带来的两个不同的例外情况给我们敲响了警钟，告诉我们（对应“可交换性原则”）一定需要有合适的补偿性操作来隐藏这个不同。

作者注：尽管如此，这里仍然有一个问题。例如，根据图 6.2 中给出的样本值，假设我们尝试向 S 插入关于供应商 S5 的数组，如果在新数组中的城市值是 Athens（雅典）之外的其他值，那么这个操作就会违反在 SC 上的键约束。但是只能看到连接视图 S 的用户根本不知道关系变量 SC 的存在。而与此同时，我们却并不想禁止对 S 的所有 INSERT 操作。也许这就是那种操作被拒绝而原因只是“因为系统就是这么说的”，没有进一步说明的情况（？）。而当你试图在 S 中改变供应商 S1 到 S5 的供应商号码的时候，类似的情况同样也会出现。

现在注意观察，可以发现上面所讲的对于 S 的 INSERT 规则与上一节中示例 1（那个严格的一对一案例）相应的规则是一致的。更进一步观察，会发现就如在本章开头介绍中所说的（规则部分的英文），这个规则的特征可以被比较通俗地归纳为如下描述：“在向 A JOIN B 插入数组 t 的时候，将 t 的 A 部分插入到 A 中，除非 A 中已经存在该元素，同样将 t 的 B 部分插入到 B 中，除非 B 中已经存在该元素。”

[4] 如果 t 已经存在于 S 中，那么它们都可能是空操作，但是请回忆一下，我们在第 4 章中讲过，我们并不需要去讨论这种情况。

注意：当然，说到“t 的 A 部分”，我指的意思是 t 的子数组，它是基于 A 的属性的 t 的投影（数组），同理适用于“t 的 B 部分”。对于“已经存在该元素”，“该元素”指的是 t 的 A 部分或者 B 部分，而整句的意思是指在相应的关系变量（A 或 B）中已经有了等价于相应子数组的数组存在。

对于S的INSERT操作到目前就是这样。那么关于DELETE操作呢？首先我要重申，我们现在所讨论的例子是关于等效于ST{SNO} = SC{SNO}的约束没有被明确声明的情况。当然，这个约束没有被声明也并不意味着它不应当起效！它可能起效，也可能不起效。让我们换个方式来说明这点，如果数据库管理系统不知道该约束应当起效，这并不意味着它知道它不该起效（这里大家跟得上吗）[5]。那么问题变成了：数据库管理系统应当做什么呢？下面有两个假设：（a）应当假设该约束无论怎样都起效呢，还是（b）应当假设该约束无论怎样都不起效呢？（当然不用说的是数据库管理系统必须基于这两种假设之一来运作。）那么我们就来看一看，如果数据库管理系统基于假设（a）来运作，那么它会尽其所能来保证所有供应商都能够同时在ST和SC中表达（当然前提是它们能够被表达），即使一个供应商“不是那么能够被表达”也不会造成违反黄金法则的结果。（不会造成违反黄金法则的结果恰恰是因为该约束没有被声明出来。）实际上，如果数据库管理系统确实基于假设（a）来运作，那么更新规则，特别是删除规则，将会变为与那些严格的一对一连接的情况完全相同，这就是结论。那么如果数据库管理系统按照假设（b）来运作的话，又会出现什么情况呢？

在我试着回答这个问题之前，先让我在这里说明，假设（b）并不是对应着一个约束，而是对应“没有”一个约束。我这样说是希望大家都能明白，我们无法直接写出一个正式的 CONSTRAINT 声明，里面说“某个供应商号码出现在 ST 中但不在 SC 中”是合法的，或者用其他方式来达到这个效果。（我在这里提到这一点是因为实际上我遇到过有人显然相信与我的说法完全相反的情况。如果你也是这样，那么我建议你试着去写出这样一个 CONSTRAINT 声明。如果你尝试一下，我相信你会很快发现我说的是正确的——如果你现在还没有被说服的话。）

那么接下来，就让我们来仔细看看假设（b）吧。我们先来考虑一个实际的例子。假设我们尝试从 S 中删除数组（S1，20，London）。显然，数据库管理系统可以通过执行下列几条操作之一来达到我们想要的效果：1. 从 ST 中删除数组（S1，20），

[5] 用符号表示就是：NOT(knows(p)) doesn’t imply knows(NOT(p))。或者如其他人所说：“证据缺失不是缺失的证据。”这里需要提醒你的是如果一个特定的约束应当起效，但是数据库管理系统却没有被通知到这一点，那么这个设计就是不完整的，并且一定会出现各种各样的错误。但是这种违反优秀设计准则的情况在实际案例中并不是没出现过。

2. 从 SC 中删除数组（S1，London），3. 前两条都执行。换句话说，我们可以认为下面几条都可以用来作为 S 的删除规则。

1．仅仅级联删除到 ST。

2．仅仅级联删除到 SC。

3．删除同时级联到 ST 和 SC。

> 当然这里还有第 4 种可能，那就是：当我们不知道应当从这 3 个选项中选择谁的时候，可以拒绝所有删除请求。老实说，一般来说让用户要求的合理操作请求成功执行是一个令人满意的结果，但是，看起来接受删除操作并让它做相应的级联这种方式更好，特别是如果假设（a）是真正正确的选择，也就是说即使没有相应的约束被显式地声明，ST{SNO} 和 SC{SNO}依然是等价的，那么级联（具体来说就是选项 3，删除同时级联到 ST 和 SC 上）确实是逻辑上正确的步骤。

现在，我确定你已经意识到了，我们正面临一个大问题。没错，现在我们正在讨论的就是经典的歧义问题，或者至少它被某些人描述为歧义问题。这个问题的重点在于这里似乎并没有一个太好的办法从众多的选项中选出最好的一项，而批评家们所抱怨的也正是这点。这也恰恰是为什么某些作者相信一般情况下视图更新是不可行的。所以在这里我想提出我对该争议的观点，这也是从我自己的角度来看比较好的选择，那就是我想我们应当选择第 3 项，即同时级联到 ST 和 SC。不过在此之前，请允许我指出，目前我们正在讨论的连接案例的投影中并不会出现类似的问题，尽管我在前面说过，连接视图和投影视图是同一硬币的两面。接下来就说明一下我的观点，选择第 3 项的原因是这样的：如果我们的流程从关系变量 S 开始，并用其投影 ST 和 SC 来代替其本身的话，那么显然任何可以用第 1 种设计表达的信息同样也都可以通过第 2 种设计来表达。但是如果我们的流程从 ST 和 SC 开始，并用其连接 S 来代替它们本身的话，那么能够被原始设计表达的部分信息很有可能无法通过替代的设计来表达，这时候信息等价就会消失。而只有当信息等价消失的时候，歧义问题才会出现。

## 6.2.1 编译指示

如我所说，我会对该争议提出我所支持的观点，那就是我们应当选择第 3 项，换句话说就是我们应当尽可能地像对示例 1 一样对待示例 2。不过首先让我先来给大家回顾一下第 3 章中所讲过的内容。

即使当信息等价不成立的时候，我们有时也能够应用视图更新规则（可能只能部分应用），那么我也会详细地讨论这种情况。**但我必须强调，在这种情况下实施的更新可能导致如下结果，就是更新操作所应具有的可预测性和清晰的定义，有时可能无法具备，甚至可能是不可接受的。**正因如此，我会给大家（可能是某个用户，可能是数据库管理员，甚至可能是数据库管理系统）留出在这种情况下对是否允许更新操作进行选择的机会。

我们现在手里的例子就是上面这段话的完美例证，而选择第 3 项意味着将信息等价情况下使用的规则应用到信息隐藏或消失的情况中。换句话说就是在我的观点中有一个清晰的编译指示元素被引入了[6]。那么现在，我将尽我所能来说服你这个编译指示是合理的，特别是，正如从第 3 章所引用的，至少它能够产生出具有的可预测性和清晰的定义的结果。但是，这个引用同样也说明了，对于这个具体的问题，很有可能其他人会不认可我的观点，因此想要选择与我所建议的不同的选项。我会在接下来的这一节的结尾重新来讨论一下这个可能性。不过首先，让我试着把我的观点解释清楚。

### 6.2.2 对称

我相信对称一般来说是一个很好的设计原则。下面是对 Polya 一些文章的引用。

- “如果一个问题从某个角度来说是具有对称性的，那么我们可能能够通过发现那些可交换的部分来得到一些启发和提示，而用相同模式对待那些扮演相同角色的元素则通常是需要付出代价的……试着用对称的方式去对待那些对称的事物，并且不要肆意破坏任何自然形成的对称”（George Polya：《How To Solve It》第 2 版，普林斯顿大学出版社，1971 年）。

- “我们期待能从数据中找到任何对称，这样问题的状况将会通过解决方案被镜像出来……对称应从对称中产生”（George Polya：《Mathematical Discovery: On Understanding, Learning, and Teaching Problem Solving》第 2 版，John Wiley & Sons，1981 年）。

作者注：Polya 也在他晚些时候的另一本书中发声，书名是《The Principle of Nonsufficient Reason》，他说：“如果没有足够的理由从一堆选项中做选择的话，那么没有任何一个[方案]应当受到特别青睐。”也许有些人会站出来说这个原则相当于拒绝了所有在连接视图中的删除操作，至少就目前我们遇到的这个连接视图来看。不过在我看来，这个观点太极端了，有点讳疾忌医的感觉。接下来，还是让我继续阐述能够让我们不走极端，而是直接选择第 3 项的原因吧（我认为它确实是个好原因）。

[6] 在本书中，我选择忽略一个事实，那就是我所查过的所有词典中没有一个把 pragma（编译指示）认作一个正经的英文词汇。

在我的经验中，对称通常意味着我们把哪里搞错了。至少，它会导致一些不太正常的行为出现，看起来变化多端让人很难以捉摸，也正因如此，它难于理解、教授、学习和记忆。那么就目前的例子而言，对称的解决方案就意味着将 S 上的删除操作仅仅级联到 ST 而没有 SC（选项 1）或者反过来（选项 2）。这个随意性的解决方案至少会导致两个很不好的结果。第一，这意味着系统管理员，或者其他的一些系统管理人员必须介入进来在两个选项中做出选择，而通常来讲并没有什么好的指导方针或者案例能够帮助他们做出选择。第二，这意味着定义为 ST JOIN SC 的视图和定义为 SC JOIN ST 的视图有可能出现不同的更新动作，而这恰恰是大家不希望得到的结果。因此我认为单独就对称的考虑已经足以让我们弃用选项 1 和选项 2 了。

我进一步观察发现，如果假设（a）是正确的话，那么选项 1 和选项 2 实际上就是错误的选择。也就是说，只有当明确了一个供应商可以拥有状态但没有城市，或者只有城市没有状态的情况下，选项 1、选项 2 以及假设（b）才能作为安全的办法可选。但在我们只知道 S = ST JOIN SC 的情况下，数据库管理系统并不知道，也不可能知道是否每个变量都符合上述情况。而如果假设（b）实际成立的话，至少选项 3 仍然有效，并且它可以产生一个可预测的结果。

因此让我们来考虑一下选项 3。显然，这个选项具有一个独特的优势，它避免了上述问题的出现。不过这里还有一个原因令这个选项颇具吸引力，实际上是另一个对对称的引申，那就是，在选项 3 的情况下，删除规则对于其相对的插入规则来说是对称的。（我们已经看到了在S上的插入操作会同时级联到ST和SC，那么为什么在S上的删除操作不能用类似的方式呢？）实际上，大体上我们可以说选项 3 提供了下面几点：（a）从S上删除一个存在的数组t，然后再把它插入，（b）向S插入一个新数组t，然后再把它删除，二者都是保持现状不变[7]，这样看起来既合理，又令人满意。

作者注：不过，我还是需要将上面的问题说得更详细一些。当我说“保持现状不变”时，我指的是针对关系变量 S（那个连接视图）来说状态不变，并且我所说的百分之百正确。而对于关系变量 ST 和 SC 来说，很不幸状况就要复杂一些。具体来说就是：对于这两个关系变量而言，会出现下面两种情况：（a）从 S 中删除已经存在的数组，然后再插入，但是在以下条件下则不一定；（b）一个新数组插入到 S 中然后再删掉。在（a）的情况下可以实现状态不变，而对（b）来说则不一定。出现这种差异的原因是向 S 中插入一个新数组实际上可能导致该数组被插入到 ST 和 SC 其一之中，反之，在选项 3 的情况下，从 S 中删除一个存在的数组总是会导致该数组从 ST 和 SC 两者中都被删除。

[7] 选项 1 和选项 2 也是一样。

这里还有一个有关系的问题。假设我们开始的时候仅有关系变量 ST 和 SC（也就是没有连接视图），令供应商 S1 同时出现在 ST 和 SC 中。那么很显然，我们可以从关系变量 ST 中删除供应商 S1，而并不用级联到关系变量 SC 中。但是假设现在我们将连接视图 S 介入进来。那么在选项 3 的情况下，该删除操作就会级联到关系变量 SC，此外，即使连接视图 S 对发起在 ST 上的删除的用户不可见，系统依然会这样执行！在后面第 9～11 章，以及第 15 章中，我还会回过头来针对这个问题以及一些相似的问题做进一步的讨论。注完。

另外还有一个支持选项 3 的原因是：如果我们同意这个选项，那么我们就会拥有一个单独的、统一的规则可以广泛应用于所有连接的删除操作，不管这个连接是一对一的、一对多的，还是多对多的。当然，我现在还没有论证这个观点，不过在后面两章我将会对其进行证明。

所以我的结论是：我认为让数据库管理系统在假设（a）的情况下工作比较好，即使相应的约束没有显式地声明出来也是一样。换句话说，我认为在上一节中为严格的一对一连接所给出的视图更新规则应当被继续遵循，即使这个连接并不是严格意义上的一对一。那么现在我来说明，正如在本章介绍中所说的，选项 3 删除规则大体上可以被直观地概括为以下形式："当从 A JOIN B 中删除数组 t 时，从 A 中删除 t 的 A 部分，从 B 中删除 t 的 B 部分"，我们还可以对这个规则的每个分句加上一句"除非它存在于该关系变量以外的其他地方"，因为我们在讨论一个一对一的连接，所以理论上 t 的 A 部分就不会存在于 A 以外的其他地方，对 B 的部分也是一样。

如果我们能够认同上述的观点，那么我们就等于认可了一套通用的一对一连接视图更新规则，而这套规则至少总能起效，并且当这些连接应当是一对一时可以保证它们一定是。除此之外，即便这些规则有时会给出一些不太好的结果，那么也总会有特定的临时补救措施（诸如使用数据库管理系统授权的子系统来禁用某些特定的更新等措施）可以采用，从而避免这些结果。

在这里请一定不要误解我的意思。我并没有说为了系统正常运行我们必须采用这类补救措施。如果一个系统只能完全依靠用户或数据库管理员的正确操作，让他们总是"做正确的事"，比如正确使用授权子系统，这显然是不可接受的[8]。所以我们必须至少总是允许

[8] 这里正好说明一个更常规些的观点，那就是无论我们采用了什么样的视图更新规则，它们都不能指望"数据库设计良好"这一点（比如，它们不能假设关系变量永远是正常的状态）。换句话说，我们必须给出数据库设计者一些空间来犯错。注意：关于这一点，我觉得也是不得不说的，那就是通常来讲，很多复杂情况伴随视图更新一同出现恰恰是在数据库设计不好的时候，特别是当数据库不完整的时候，这种情况下，一些应该捕捉到的因素就无法被捕捉到。

连接视图更新，即使这些连接不是严格的一对一形式，而且我们还需要一套在这种情况下也能起效的规则。这就是为什么我提倡我的观点[9]。

## 6.3 小结

在这里，有关一对一的连接，我还有最后几点想要阐述。第一点是这样的。关于我们所讲的例子，我已经考虑了下面两种可能性：每一个供应商都既有状态值，又有城市值，或者是，每个供应商都至少拥有其中一种属性值。但是，这里至少还有两种不同的可能性。

- 这两个属性具有相互排斥性，即每个供应商都只能拥有其中一种属性。
- 其中一个属性是强制绑定的，而另一种则是可有可无（比如，所有的供应商都有城市值，而其中的一些同时也拥有状态值）。

其实只需要看一眼就足以判断出第一种可能性没什么意思。原因就在于这种供应商总是能被 ST 和 SC 的其中之一表达，而永远不可能被两个关系变量同时表达。因此连接 S 永远都为空，并且对该连接的删除操作永远都是“空操作”，而对该连接的插入操作则永远都会失败（除非插入也是“空操作”）。至于第 2 个可能性，我认为它也许可以被证明是适合选项 1 或选项 2 删除规则的一种情况。也许是吧，我并没有找到一个能够说服自己的角度，不过我认为我应当是能被说服的，否则我就会在这里大声疾呼要求证明了。不过就目前来讲，我姑且还是按照我在本章早些时候所说的那样，保留我的像“万能钥匙”般的规则吧，它当然依旧有效，即使是在当前我们所讨论的这样特殊的案例中也是一样。

另外一点我要说的，当然也是广为人知的一点，就是交集实际上是一对一连接的一种特殊形式[10]。由此我们可以得出，通过一对一连接进行更新的规则同样适用于交集（更准确一点说就是规则缩小到通过交集更新的范围，而此时的连接本身则缩减为特殊的“交集”形式）。下面我们就来说一说这些规则吧（虽然使用比较精简的

[9] 如果你不喜欢你做了某事后产生的结果，那么你就不要去做某事——“如果你觉得用锤子敲自己的头很疼，那么就不要用锤子敲自己的头”，这个观点有时候指的是 Groucho 原则。在这里我的建议（也就是，如果你对执行某项更新的结果不满意，那就不要执行该项更新）可以看作是对该原则的引用。那么一般来讲，我很赞同这一点，那就是如果对你的观点只能找到 Groucho 原则来做支撑，那么你的观点一定特别脆弱。当然，我并不相信对于我们手上的例子来说，只有 Groucho 原则来做支撑。

[10] 这就是严格与否的区别，虽然实际上“严格”在这里的意义不大。（为什么？）

形式），[11] 令V定义为A INTERSECT B。于是我们就会有：

```
ON INSERT i INTO A :
   INSERT ( i INTERSECT B ) INTO V ;

ON INSERT i INTO B :
   INSERT ( i INTERSECT A ) INTO V ;

ON DELETE d FROM A :
   DELETE d FROM V ;

ON DELETE d FROM B :
   DELETE d FROM V ;

ON INSERT i INTO V :
   INSERT i INTO A ,
   INSERT i INTO B ;

ON DELETE d FROM V :
   DELETE d FROM A ,
   DELETE d FROM B ;
```

现在看来，我要有更多关于交集的东西要在第 9 章讲了，不过我曾想过要在这里谈一谈，因为这些内容可以让我将本章前面所提到过的“经典的歧义问题”用逻辑术语表述出来。我前面曾经说过，在“一对一”这个术语使用得不太严格的情况下，比如在示例 2 中，通过一个一对一连接进行删除操作的规则需要引入特定程度的编译指示。那么现在，我则可以敲定到底这个编译指示都包括什么内容。请大家回想一下第 2 章，每一个关系变量，或者更进一步说是每一个关系表达式都拥有一个与之相对应的谓词（在接下来的例子中，谓词是由表达式所涉及的关系变量的谓词，根据该表达式所涉及到的关系操作的语义所推导出来的）。那么令A和B的谓词分别为PA和PB，于是V = A INTERSECT B的谓词就是PA AND PB。这就是说，如果数组t在V中出现，那么PA(t) AND PB(t)的值为真（这当然也就是说PA(t)和PB(t)各自本身均为真）。[12]

那么现在假设我们要从 V 中删除 t。我们进行这个更新的目的是让命题 PA(t) AND PB(t)不再为真，换句话说就是 PA(t)与 PB(t)之中至少有一个是假的，也有可能两个同时为假。对比来说，从 A 和 B 中都删除 t 的意思是“PA(t)为假 AND PB(t)为假”，换句话说就是都为假。于是接下来我们便有了歧义问题的逻辑描述，也就是，我所提出的选项 3 在“一对一”这个术语使用得不太严格的情况下，通过一个一对一连

[11] 在第 9 章，我将会对这些规则做一些改进。

[12] 我用表达式 PA(t)来指出通过将谓词 PA 实例化所得到的命题，方法是用数组 t 的属性值作为参数来替换 PA 中的参数，对于 PB(t)来说方法类似。

接进行删除操作的规则，相当于把逻辑 OR 当作逻辑 AND 对待。

我们或许也会说从 V 中删除 t 意思是 NOT(PA(t) AND PB(t))为真，而从 A 和 B 中都删除 t 则意为 NOT(PA(t)) AND NOT (PB(t))为真。但是 NOT(PA(t)) AND NOT(PB(t))等价于 NOT(PA(t) OR PB(t))，因此选项 3 的删除规则意味着我们事实上相当于将逻辑 AND 当作逻辑 OR 对待（换句话说就是从反方向来说）。当然，那种方式对我们的目标来说都没有不同。

下面我不会在这里说更多关于这个问题的内容了，我将会在第 15 章为大家介绍 David McGoveran 的一个方案来解决这个问题。

第 7 章

# 连接视图 II：多对多连接

最近的研究在哈佛，更多研究在耶鲁，展示了连接可以更新，只要把尾巴上的钉子取出即可。

——Anon：《Where Bugs Go》

下面我想检验一下多对多连接的问题（我故意把一对多的情况放到最后再讲）。正如在第 6 章中所说的，我想在展开接下来的调查之前进行说明会对大家有些帮助。大家可能会预见到，多对多的情况将会引发特定的“并发症”，而这些“并发症”在一对一的情况下并不会出现。尽管如此，我确定我们依然会使用本质上与之前一样的规则来解决问题。也就是说，定义视图 V 为 A JOIN B，只不过这个连接现在是个多对多连接，而规则依然看起来如下。

```
ON INSERT INTO V : INSERT A (sub)tuples if they don’t already exist,
                   INSERT B (sub)tuples if they don’t already exist

ON DELETE FROM V : DELETE A (sub)tuples if they don’t exist elsewhere,
                   DELETE B (sub)tuples if they don’t exist elsewhere
```

## 7.1 示例 1：信息等价

让我们考虑一个简化版的供应商关系变量与零部件关系变量，其中的供应商除了 SNO 和 CITY 之外不含其他属性，而零部件则除了 PNO 和 CITY 外不含其他属性，于是有：

```
S { SNO , CITY } KEY { SNO }
P { PNO , CITY } KEY { PNO }
```

为了让例子能更好地说明问题，我们同时假设每个供应商的城市要求与零部件的城市一致，反之亦然，换句话说就是这里会有一个约束起效，其实相当于又是一个相等依赖条件，形式如下：

```
CONSTRAINT ... S { CITY } = P { CITY } ;
```

为了符合这一新的要求，让我们暂时将供应商 S5（城市 Athens）的数组和零部件 P3（城市 Oslo）从以往的供应商关系变量和零部件关系变量中移除，直到后面再有安排为止。

现在，让我们来定义一下关系变量 S 和 P 的连接视图 SCP。

```
SCP { SNO , CITY , PNO } KEY { SNO , PNO }
```

通过观察这个连接我们不难发现，它确实是一个多对多的连接，在这里 S 中的每一个数组都与 P 中的许多数组连接，反之亦然。尽管如此，我们依然拥有信息等价，正是因为每个关系变量中的每一个数组都至少与另一个关系变量中的一个数组连接。样本值如图 7.1 所示。

S

| SNO | CITY |
|---|---|
| S1 | London |
| S2 | Paris |
| S3 | Paris |
| S4 | London |

P

| PNO | CITY |
|---|---|
| P1 | London |
| P2 | Paris |
| P4 | London |
| P5 | Paris |
| P6 | London |

SCP

| SNO | CITY | PNO |
|---|---|---|
| S1 | London | P1 |
| S1 | London | P4 |
| S1 | London | P6 |
| S2 | Paris | P2 |
| S2 | Paris | P5 |
| S3 | Paris | P2 |
| S3 | Paris | P5 |
| S4 | London | P1 |
| S4 | London | P4 |
| S4 | London | P6 |

图 7.1　关系变量 S、P 以及 SCP 的样本值

下面是相应变量的谓词[1]。

S: 供应商 SNO 拥有城市 CITY。

P: 零部件 PNO 拥有城市 CITY。

[1] 我已经对这些谓词做了些许简化：（a）对供应商去掉了“已经签约”，对零部件则去掉了“在公司内使用”，（b）供应商和零部件都“拥有”某城市，而不说供应商“位于”某城市，也不说零部件“存储在”某城市。当然这些变动纯属为了讲解方便，我所做的第一个变动仅仅只是为了简化，而第二个变动则是为了强调在本例中，不论是供应商，还是零部件，所扮演的都是平行的角色。

SCP: 供应商 SNO 和零部件 PNO 都拥有城市 CITY。

如同往常一样，我将从所有 3 个关系变量都是基础关系变量的情况开始分析。在这种情况下，有哪些约束是起效的呢？我们来看看，首先，这 3 个关系变量分别拥有{SNO}、{PNO}和{SNO,PNO}，作为它们各自的单键，这些我们都已经讲过。其次，我们显然拥有下列几个约束。

```
CONSTRAINT ... SCP = S JOIN P ;
CONSTRAINT ... S = SCP { SNO , CITY } ;
CONSTRAINT ... P = SCP { PNO , CITY } ;
```

当然，这些约束都是相等依赖条件（EQD），它们在一起来保证只包含关系变量 SCP 的设计，以及由关系变量 S 和 P 整合而成的设计这两种方式之间是信息等价的，这个在前面也有讲过。

下面这几条约束，同样也是 EQD，它们也会起效，实际上它们是上述 3 条约束导致的逻辑结果，详细如下。

```
CONSTRAINT ... IDENTICAL { S { CITY } , P { CITY } , SCP { CITY } } ;
CONSTRAINT ... SCP = JOIN { SCP { SNO , CITY } , SCP { PNO , CITY } } ;
```

不过我还是想详细说一下最后这条约束。它传达了这样一个事实，那就是关系变量 SCP 并不属于第四范式（4NF），因为这个约束等价于下列的连接依赖（join dependency JD）。

```
☼{ { SNO , CITY } , { PNO , CITY } }
```

而这个 JD 又与下面这对多值依赖（multivalued dependencies MVD）等价。

```
{ CITY } →→ { SNO }
{ CITY } →→ { PNO }
```

或者把它们写成“一行”。

```
{ CITY } →→ { SNO } | { PNO }
```

这些MVD并不是平凡的，也不是由SCP的单键{SNO,PNO}来表示的，因此SCP并不符合 4NF[2]。注意：这些MVD的意义我们从不那么正式的角度来解释就是如果数组（c, s, p）和（c, s’, p’）出现在该关系变量中，那么数组（$c, s, p'$）和（$c, s', p$）必须

[2] 对于一个 MVD 而言，当且仅当以下两种条件之一出现时是平凡的：（a）决定量（表达式左侧的属性集合）包含了依赖量（表达式右侧的属性集合），或者（b）决定量和依赖量的合并等于整个关系头。而一个非平凡的 MVD 当且仅当下面条件成立时可以由键来表示，决定量是一个超级键。如果你想获得关于此内容的更多解释，请参看《Database Design and Relational Theory》。

也同时出现，在这里c是城市，s和s’是供应商号码，而p和p’则是零部件编号。为了更好地描绘这一点，我们选取图 7.1 的值，并令c等于Paris，令s和s’分别等于S2 和S3，而令p和p’分别等于P2 和P5。

当然，上面所说的 MVD 出现在 SCP 中并不是偶然的，实际上只要是在多对多连接的情况下，刚刚我们看到的这些 MVD 就都会存在并起效。而我要在这里讲这些内容的原因是：MVD 所在的关系变量，即非 4NF 的关系变量，显然是受制于冗余的，而总体来说受制于冗余的关系变量很难进行更新，我们之后就会从手中的例子看到这样的现象。

在依赖这个主题上我还有些话要说，那就是下面所列的函数依赖（FD）也是存在并起效于SCP之内的[3]。

```
{ SNO } → { CITY }
{ PNO } → { CITY }
```

这些 FD 是从关系变量 S 和 P 那里分别继承来的，它们在各自原始的关系变量中显然是存在并起效的，因为决定量，也就是表达式左侧的属性，是它们各自关系变量的键。

### 7.1.1 补偿性操作

现在我们来看下面这样一个在关系变量 S 上的 INSERT 操作。

```
INSERT ( S9 , London ) INTO S ;
```

代入图 7.1 的样本值，这个插入操作可以并且将会成功，而它同时拥有一点额外的效果就是将下列几个数组插入到关系变量 SCP 中。

```
( S9 , London , P1 )
( S9 , London , P4 )
( S9 , London , P6 )
```

让我们再来对比下面的这个 INSERT 看看。

```
INSERT ( S9 , Madrid ) INTO S ;
```

同样代入图 7.1 的样本值，这个插入操作一定失败，因为在 Madrid 没有任何零部件，而在这里补偿性操作也不可能有意义。另一方面，下面这个双 INSERT 操作就可以并且将会成功。

```
INSERT ( S9 , Madrid ) INTO S ,
INSERT ( P8 , Madrid ) INTO P ;
```

[3] 这样的后果是 SCP 甚至都不能算是第二范式（2NF），更不用说第四范式了。但正是这种非 4NF 恰恰导致了关系变量具备了多对多连接的特性。

同样，它也具有额外的效果，把下列数组插入到关系变量 SCP 中。

```
( S9 , Madrid , P8 )
```

通过我们这里所讲的这些例子以及其他与之类似的案例，我希望大家已经能够清晰地看出下面我们列出的就是一个对于在 S 和/或 P 上很适合的插入规则。

```
ON INSERT is INTO S , INSERT ip INTO P :
   INSERT ( P JOIN is ) INTO SCP ,
   INSERT ( S JOIN ip ) INTO SCP ;
```

那么下面我们再来看一看删除操作。假设就在现在，我们暂时恢复关于供应商 S5 和零部件 P3 的数组，S5 的城市也如同往常一样是 Athens，不过我们将 P3 的城市从 Oslo 改为 Athens。于是，关系变量 SCP 额外地包含了数组（S5，Athens，P3），不过请注意，这个数组是该关系变量中唯一的含有 Athens 这个城市的数组。那么现在，下面的双 DELETE 显然是没问题的。

```
DELETE ( S5 , Athens ) FROM S ,
DELETE ( P3 , Athens ) FROM P ;
```

它们的效果是从 S 和 P 中删除这两个具体的数组，并且在相应的级联删除规则起效的前提下，同时也从关系变量 SCP 中删除数组（S5，Athens，P3）。那么对于下面的这个单一 DELETE 又会怎样呢？具体如下。

```
DELETE ( S5 , Athens ) FROM S ;
```

现在，这里不能被允许发生的就是与需求的完全一致，从关系变量 S 中删除数组（S5，Athens），而其他一切都不变。而这违反了黄金原则，因此这个 DELETE 操作就会被直接拒绝。通常情况下，让用户的合理需求得到满足是非常理想的结果，但是，看起来接受 DELETE 操作并让它做出相应的级联在这里是更好的选择。换句话说，我认为下面所列的是对于关系变量 S 和/或 P 的一个比较合适的删除规则。

```
ON DELETE ds FROM S , DELETE dp FROM P :
   DELETE ( SCP JOIN ds ) FROM SCP ,
   DELETE ( SCP JOIN dp ) FROM SCP ;
```

有了上面的插入和删除规则，我就把验证“在 S 和/或 P 上的显式 UPDATE 操作都能正常执行”的工作留给大家作为练习了。

下面我们来谈谈在连接 SCP 上的更新情况。插入规则还是很明显的。

```
ON INSERT i INTO SCP :
```

```
INSERT i { SNO , CITY } INTO S ,
INSERT i { PNO , CITY } INTO P ;
```

我确实说这个规则很明显，但是它所导致的结果却并不一定这么明显，至少不会马上就显现出来。让我们来看几个例子吧，样本值还是使用图 7.1 给定的值。

1. 假设我们向 SCP 插入（S9，London，P1）。这个插入操作会导致（S9，London）被插入 S，但是对 P 没有影响，因为（P1，London）已经存在于 P 中。但是向 S 中插入（S9，London）会导致 S 的插入规则起效，而最后的效果则是除了原始需求的数组（S9，London，P1）外，（S9，London，P4）和（S9，London，P6）将被额外地插入到 SCP 中。

2. 假设我们向 SCP 中插入（S7，Paris，P7）。那么最后的效果就会是向 S 中插入（S7，Paris），向 P 中插入（P7，Paris），而所有下面所列的这些数组将被插入到 SCP 中。

```
( S7 , Paris , P7 ) ( S7 , Paris , P2 ) ( S2 , Paris , P7 )
                    ( S7 , Paris , P5 ) ( S3 , Paris , P7 )
```

如果大家现在还没有意识到，我要指出的是上面所说的插入规则本质上同我们在第 6 章所讲的一对一的案例的插入规则是一样的，而在这个例子中稍有差别的是，原始的 INSERT 级联会导致向连接中插入额外的数组。

那么现在看来，上面的插入规则可以大体描述为："插入 S 子数组除非它们已经存在，插入 P 子数组除非它们已经存在。"直觉和对称都告诉我们，相应的删除规则应该是："删除 S 子数组除非它们已经在别处存在，删除 P 子数组除非它们已经在别处存在。"正式的定义如下。

```
ON DELETE d FROM SCP :
   DELETE ( ( S MATCHING d ) NOT MATCHING SCP ) FROM S ,
   DELETE ( ( P MATCHING d ) NOT MATCHING SCP ) FROM P ;
```

让我们再来看一些例子，还是使用图 7.1 的样本值。

1. 假设我们从 SCP 中删除所有城市为 Paris 的数组。这个删除操作将会级联删除关系变量 S 中供应商 S2 和 S3 的相关数组，以及关系变量 P 中零部件 P2 和 P5 的相关数组。

2. 假设我们从 SCP 中删除所有供应商 S1 的数组。这个删除操作将会级联删除关系变量 S 中供应商 S1 的相关数组，但是对关系变量 P 没有任何影响，因为 SCP 仍

然包含一些城市为 London 的数组，具体说来就是数组（S4，London，P1）、（S4，London，P4），以及（S4，London，P6）。

3. 假设我们尝试从 SCP 中只删除数组（S1，London，P1）。这个尝试一定会失败，因为 SCP 还包含其他既关于供应商 S1，又关于零部件 P1 的数组，而所尝试的删除操作对关系变量 S 和 P 没有效果，如果这个操作想要成功，那就必定违反黄金法则，具体来说就是 SCP 不再等价于 S 和 P 的连接。

那么现在，如果你把上述的删除规则与第 6 章中那个一对一连接使用的删除规则做个比较的话，会有什么结果呢？为了方便对比我把第 6 章的规则摘录如下。

```
ON DELETE d FROM S :
   DELETE d { SNO , STATUS } FROM ST ,
   DELETE d { SNO , CITY } FROM SC ;
```

你可能会觉得它们二者是有些不同的。但实际上并没有。让我们用下面的办法来看看是为什么，假设现在暂时每个城市只对应一个供应商或者一个零部件（因此多对多的连接降级为一对一的连接）。那么，我们用 s、c 和 p 来分别表示供应商号码、城市名称和零部件号码，所有（s，c）子数组都只在 SPC 中出现一次，所有（p，c）子数组也都只在 SCP 中出现一次，那么(S MATCHING d) NOT MATCHING SCP 和 (P MATCHING d) NOT MATCHING SCP 分别降级为 S MATCHING d 和 P MATCHING d。同时，S MATCHING d 和 P MATCHING d 则分别等价于 d{SNO, CITY}和 d{PNO,CITY}。因此，一对一连接的删除规则可以理解为从多对多的删除规则简化或者降级而来。

最后，根据上面的插入和删除规则来验证显式 UPDATE 全都可以正常执行的工作，就留给大家作为练习了。

### 7.1.2 视图更新

现在我们假设 SCP 实际上是个视图，是 S 和 P 的连接。那么：

1. 虽然不是全部，但绝大部分在本节开头列举的约束都会自动起效。当然，那些让投影 S{CITY}和 P{CITY}等价的约束不会自动起效。

2. 从 S 和 P 到 SCP 的补偿性操作将会自动发生，这正是因为 SCP 是一个视图，也就是说，在 S 和/或 P 上的更新会自动映射到 SCP 相应的部分。

3. 从 SCP 到 S 和 SP 的补偿性操作，也就是视图更新规则，也将会自动发生，同样也因为 SCP 是视图。也就是说，在 SCP 上的更新才是“真正”对底层关系变量 S 和/或 P 的更新，它在 SCP 中自动可见，就如同其在 S 和 P 中一样。

那么在用户只能看到连接视图 SCP 的情况下会怎么样呢？在这种情况下，该视图将可以执行所有操作，就好像它就是一个基础关系变量一样，不过需要说明的是它的操作执行起来并不都是直接进行的，我后面很快就会做出解释。这类用户将会知道{SNO,PNO}是键，同样也会知道下列 FD 和 MVD。

```
{ SNO } → { CITY }
{ PNO } → { CITY }

{ CITY } →→ { SNO } | { PNO }
```

关于这些 MVD，我要提醒你的是它们共同作用的时候等价于下面所列的约束。

```
CONSTRAINT ... SCP = JOIN { SCP { SNO , CITY } , SCP { PNO , CITY } } ;
```

该用户也将会知道相应的谓词。

SCP：供应商 SNO 和零部件 PNO 同时都拥有城市 CITY。

我之前说过SCP并不符合 4NF，而且我还说它也可能因此很难被更新（这里我要强调不论SCP是基础关系变量，还是视图，我的观点都一样）。现在让我来更具体地解释一下。首先，更新通常都必须遵守那些MVD。其次，尤其是INSERT操作受制于下列规则[4]。

```
ON INSERT i INTO SCP :
   INSERT ( SCP JOIN i { SNO , CITY } ) INTO SCP ,
   INSERT ( SCP JOIN i { PNO , CITY } ) INTO SCP ;
```

这个规则可能看起来有点复杂，但是它基本上就是之前对于 S、P 和 SCP 的 INSERT 规则的一个整合，重新调整来去掉对 S 和 P 的引用。注意观察，这意味着即使在 SCP 是基础关系变量，而关系变量 S 和 P 根本不存在（或被隐藏）的情况下，该规则应该并且确实也适用。实际上，我们本可以仅仅通过单独考虑关系变量 SCP 就能得出这个规则。同时，值得注意的是这个规则其实是一个例子，在本书中到目前为止我们所见到的第一个例子，来展示一个将特定更新级联到目标关系变量自己身上的规则。

[4] 注意观察，我并没有给出一个同等的删除规则。实际上，除非用户要求（不管是显式，还是隐式）要将某个特定的供应商所有的数组都删除，或者将某个特定零部件的所有数组都删除，否则在 SCP 上的 DELETE 操作永远都会失败。例如，按照图 7.1 的样本来说，一个需求仅仅要删除(S1,London,P1)这一个数组将会失败，而如果要求将所有城市为 Paris 的数组都删掉就会成功（如同我们在前面两个例子中看到的）。

## 7.2 再谈投影视图

我们已经讨论了当 S 和 P 为基础关系变量，而 SCP 是视图的情况。为了完整地检验示例 1，我确实应当再来讨论一下相反的情况，那就是 SCP 是基础关系变量，而 S 和 P 都是该关系变量的视图。显然，在这种情况下 S 和 P 其实就成为了投影视图，而这个讨论从逻辑上讲应该属于第 5 章的范畴。不过不管怎样，我觉得第 5 章本身已经够复杂了，因此我决定还是把这个讨论推迟到这一章来讲。

实际上我认为这里没有太多可说的。让我们回忆一下第 5 章的内容，如果 DB1 只包含一个单独的关系变量 R，而 DB2 只包含 R 的投影，那么 DB1 和 DB2 就是信息等价的，前提是这些投影是按照在 R 中起效的连接依赖（JD）分解 R 所得来的。下面这个 JD 确实在 SCP 中起效。

```
☼{ { SNO , CITY } , { PNO , CITY } }
```

由它导致的直接后果就是 SCP 可以被无损分解为关于{SNO,CITY}和{PNO,CITY}的投影，换句话说就是分解为关系变量 S 和 P，而这些关系变量必然是可更新的（只要 SCP 自己首先是可更新的，当然这已经是事实）。至于各种各样的补偿性操作都跟之前讨论的一样。

```
ON INSERT i INTO SCP :
   INSERT i { SNO , CITY } INTO S ,
   INSERT i { PNO , CITY } INTO P ;

ON DELETE d FROM SCP :
   DELETE ( ( S MATCHING d ) NOT MATCHING SCP ) FROM S ,
   DELETE ( ( P MATCHING d ) NOT MATCHING SCP ) FROM P ;

ON INSERT is INTO S , INSERT ip INTO P :
   INSERT ( P JOIN is ) INTO SCP ,
   INSERT ( S JOIN ip ) INTO SCP ;

ON DELETE ds FROM S , DELETE dp FROM P :
   DELETE ( SCP JOIN ds ) FROM SCP ,
   DELETE ( SCP JOIN dp ) FROM SCP ;
```

请允许我在这里加一句，视图 S 和 P 之间的关系是多对多。虽然我并没有在第 5 章中就这个具体例子进行探讨，但是我确实对那一章中的另一个关于在多对多关系中投影视图的例子进行了讨论，也就是示例 3，它涉及到视图 ST（S 关于 SNO 和 STATIS 的投影）以及 TC（S 关于 STATUS 和 CITY 的投影）。要说明的是那个例子中的关系确

实是多对多，因为同一个 STATUS 值可以与许多供应商和许多城市关联（在带入我们常用的样本值时，STATUS 值 30 就是个例子）。只可惜，那个例子所涉及的是一个有损分解，而不是无损的。所以，在投影关系变量上的更新并不总是进行得那么好。

## 7.3 示例 2：信息隐藏

现在让我们回到 S 和 P 都是基础关系变量的情况。假设现在，在我们最原始的供应商和零部件数据库中，一个城市可以在基础关系变量 S 中表达，但是不能在基础关系变量 P 中或通过其他方式表达，换句话说就是有些城市拥有供应商，但没有零部件，有些是反过来的情况。于是 S 和 P 的连接 SCP 出现了信息丢失，因为系统无法再保证 S 和 P 与该连接关于相应属性的投影是等价的。实际上，对于前面部分提到过的约束来说，只有下列的几个还依然适用（而这次我使用了文字描述的方式，而不是正式的语法模式）。

- 以往的键约束都会起效：{SNO}是 S 的键，{PNO}是 P 的键，{SNO,PNO}是 SCP 的键。
- 在任意给定时间，SCP 的当前值都等于 S 和 P 当前值的连接。
- FD：{SNO} → {CITY}和{PNO} → {CITY}以及 MVD：{CITY} →→ {SNO}和{CITY} →→ {PNO}都在 SCP 中起效。

（特定的包含依赖也会起效，不过正如第 6 章中示例 2 中的注解所讲的，我不会在这里多做说明。）

我们可以通过扩展图 7.1 的样本来恢复原来供应商 S5 和零部件 P3 的数组，从而获得一个数据库值能够满足上述条件的实际例子（请参看图 7.2，并且请特别注意观察，关系变量 SCP 在该图所示值的关系情况，与其在图 7.1 所示值时是一样的）。

S

| SNO | CITY |
|---|---|
| S1 | London |
| S2 | Paris |
| S3 | Paris |
| S4 | London |
| S5 | Athens |

P

| PNO | CITY |
|---|---|
| P1 | London |
| P2 | Paris |
| P3 | Oslo |
| P4 | London |
| P5 | Paris |
| P6 | London |

SCP

| SNO | CITY | PNO |
|---|---|---|
| S1 | London | P1 |
| S1 | London | P4 |
| S1 | London | P6 |
| S2 | Paris | P2 |
| S2 | Paris | P5 |
| S3 | Paris | P2 |
| S3 | Paris | P5 |
| S4 | London | P1 |
| S4 | London | P4 |
| S4 | London | P6 |

图 7.2　图 7.1 的重置版

关于术语的注释：本章的标题展示了，或者说至少是强烈暗示了 S JOIN P 依然是一个多对多连接，但是你必须理解这里所说的“多”是包含零的情况的。事实情况是术语“多对多”在使用上经常不是很严格，包含以下两种情况：（a）所谓的严格符合的情况，比如我们之前讨论的示例 1；（b）像我们目前正在讨论的这个案例，它可能出现这种情况，即一方含有某元素，而另一方却没有与之相对应的元素。下面是一段轻微编辑过的文字引自《The Relational Database Dictionary, Extended Edition》（Apress 出版，2008 年）。

严格地讲，一个多对多的对应关系是对两个集合 s1 和 s2（不是必需不同）配对的规则，它规定 s1 中的每一个元素都至少对应一个 s2 中的元素，s2 中的每一个元素也都至少对应一个 s1 中的元素。然而，这个术语却经常被使用得不那么严格，从而对以下 3 种情况进行配对：（a）s1 中的每一个元素都对应任意数量的 s2 中的元素（数量可能是 0），而 s2 中的每一个元素都至少对应一个 s1 中的元素；（b）s1 中的每一个元素都至少对应一个 s2 中的元素，而 s2 中的每一个元素都对应任意数量的 s1 中的元素（数量可能是 0）；（c）s1 中的每一个元素都对应任意数量的 s2 中的元素（数量可能是 0），而 s2 中的每一个元素都对应任意数量的 s1 中的元素（数量可能是 0）。除非想要表达的意思非常明确，否则我们最好避免使用这个术语。

虽然文中最后一句话是这么写的，但我在本章中还是偶尔会使用一下这个术语，不过我想我的指示意义在上下文之间都是很明确的。

那么接下来，我想向大家阐述的是如我之前所讲的，恰恰是因为有信息丢失，从而导致只有关系变量 SCP 的设计已经不再信息等价于由关系变量 S 和 P 整合组成的设计了（这里举一个后者中元素在前者中没有对应元素的查询例子，那就是“取得一个没有零部件的城市中的供应商号码”）。更具体点来说就是任何可以由关系变量 SCP 单独表达的信息一定可以被关系变量 S 和 P 的整合所表达，而反之则不成立。由此导致的结果就是显然这里会有特定的更新可以在 S 和/或 P 上执行，但却没办法相应地在 SCP 中进行。一个例子就是“向 S 插入一个城市为 Madrid 的供应商数组”，而没有同时向 P 中插入相同的城市。

我并不想在这里对这个例子做过多的调研分析。相反，我要直接援引第 6 章中对示例 2 的分析。更具体地讲，就是我要说在第 6 章中对那个例子的细节分析稍作修改就完全可以适用于现在这个例子。换句话说，就是我认为在本章前面为严格的多对多连接视图更新所修改过的规则，如果需要则可以直接应用于本例。改述一句我们在第 6 章中讨论的话，如果我们能够认可这个观点，那么我们至少能够获得一系列通用的多对多连接视图更新规则，这些规则总是有效，并且在多对多连接应当严格符合条件时保证它们确实符合，更不用说这个规则其实对所有连接视图都有效，不

管连接是一对一、一对多，还是多对多了。除此之外，如果这些规则有时因为某种原因偶尔会出现不好用的情况，那么这里也有特定的解决办法，比如用数据库管理系统授权的子系统来禁止某些特定的更新，这可以解决以上问题。

## 7.4 小结

关于多对多连接我还有一点想要说。那就是它有一个必要条件，相信大家也都知道，那就是笛卡尔乘积是一个多对多连接的特殊情况。由此可以得出，对于通过多对多连接更新的规则同样适用于笛卡尔乘积，更具体地说就是这些规则简化为通过笛卡尔乘积进行更新的规则，而连接本身也简化为一个笛卡尔乘积。让我们把结果列出来看看，定义 V 为 A TIMES B，令 HA 和 HB 分别为 A 和 B 的属性，并且令 HA 和 HB 没有相同的属性名。那么我们就会有：

```
ON INSERT i INTO V :
   INSERT i { HA } INTO A ,
   INSERT i { HB } INTO B ;

ON DELETE d FROM V :
   DELETE ( ( A MATCHING d ) NOT MATCHING V ) FROM A ,
   DELETE ( ( B MATCHING d ) NOT MATCHING V ) FROM B ;

ON INSERT ia INTO A , INSERT ib INTO B :
   INSERT ( B TIMES ia ) INTO V ,
   INSERT ( A TIMES ib ) INTO V ;

ON DELETE da FROM A , DELETE db FROM B :
   DELETE ( V MATCHING da ) FROM V ,
   DELETE ( V MATCHING db ) FROM V ;
```

第 8 章

# 连接视图 III：一对多连接

凡是 Codd 连接的，小心更新是没错的。

——Anon：《Where Bugs Go》

我们在第 6 章和第 7 章分别讨论了一对一和多对多两种连接方式，那么在这一章我们要讨论的，就是剩下的唯一一种方式，一对多连接。不过现在你一定知道我们调研讨论的走向是怎样的了，我们还是像之前两章一样从那些常用的规则讲起。

```
ON INSERT INTO V : INSERT A (sub)tuples if they don’t already exist,
                   INSERT B (sub)tuples if they don’t already exist

ON DELETE FROM V : DELETE A (sub)tuples if they don’t exist elsewhere,
                   DELETE B (sub)tuples if they don’t exist elsewhere
```

## 8.1 示例 1：信息等价

我还是要从我们常用的供应商-零部件数据库开始本节的内容，但是在本章中我将重点关注关系变量 S 和 SP，而忽略掉关系变量 P，我也会继续为了简化问题而忽略掉属性 SNAME 和属性 STATUS。这样，我们就拥有了两个基础关系变量，具体如下。

```
S { SNO , CITY } KEY { SNO }
SP { SNO , PNO , QTY } KEY { SNO , PNO }
```

另外，假设在示例 1 中，每个供应商都至少供应一种零部件。换句话说，我们假设这里有一个约束起效（实际上就是一个相等依赖条件），详情如下。

```
CONSTRAINT ... S { SNO } = SP { SNO } ;
```

为了符合这个要求，我们也会将供应商 S5 从本例的供应商关系中移除。

那么下面，我们将关系变量 S 和 SP 的连接定义为视图 SSP。

```
SSP { SNO , CITY , PNO , QTY } KEY { SNO , PNO }
```

注意观察，现在这个连接就是一对多的了，其中 S 中的每一个数组都与 SP 中的多个数组（至少是一个）连接，而 SP 中的每个数组都只与 S 中的一个数组连接。两个不同设计，仅由关系变量 SSP 组成的设计，和由关系变量 S 与 SP 整合组成的设计之间显然是信息等价的。样本值如图 8.1 所示。

S

| SNO | CITY |
|---|---|
| S1 | London |
| S2 | Paris |
| S3 | Paris |
| S4 | London |

SP

| SNO | PNO | QTY |
|---|---|---|
| S1 | P1 | 300 |
| S1 | P2 | 200 |
| S1 | P3 | 400 |
| S1 | P4 | 200 |
| S1 | P5 | 100 |
| S1 | P6 | 100 |
| S2 | P1 | 300 |
| S2 | P2 | 400 |
| S3 | P2 | 200 |
| S4 | P2 | 200 |
| S4 | P4 | 300 |
| S4 | P5 | 400 |

SSP

| SNO | CITY | PNO | QTY |
|---|---|---|---|
| S1 | London | P1 | 300 |
| S1 | London | P2 | 200 |
| S1 | London | P3 | 400 |
| S1 | London | P4 | 200 |
| S1 | London | P5 | 100 |
| S1 | London | P6 | 100 |
| S2 | Paris | P1 | 300 |
| S2 | Paris | P2 | 400 |
| S3 | Paris | P2 | 200 |
| S4 | London | P2 | 200 |
| S4 | London | P4 | 300 |
| S4 | London | P5 | 400 |

图 8.1　关系变量 S、SP 以及 SSP 的样本值

下面是几个变量各自的谓词。

S：供应商 SNO 拥有城市 CITY。

SP：供应商 SNO 供应零部件 PNO 数量为 QTY。

SSP：供应商 SNO 拥有城市 CITY，并提供零部件 PNO 数量为 QTY。

像往常一样，我们还是先来看看如果三者都是基础关系变量会是什么情况。如果它们是，那么什么约束会起效？首先，关系变量 S、SP 和 SSP 分别拥有{SNO}、{SNO,PNO}和{SNO,PNO}作为它们的单键（这些我们已经说过）。第二，我们显然拥有如下约束。

```
CONSTRAINT ... SSP = S JOIN SP ;
CONSTRAINT ... S = SSP { SNO , CITY } ;
CONSTRAINT ... SP = SSP { SNO , PNO , QTY } ;
```

这些约束全部都是相等依赖（EQD），而它们合在一起则能够保证前面说的信息等

价。实际上，我们可以将S和SP的整合看作对SSP的无损分解，因为下面的函数依赖（FD）在后者（SSP）中起效[1]。

```
{ SNO } → { CITY }
```

而由此，这个示例在本质上与第 5 章的示例 2 是非常相似的。正因为如此，我就不再对它相应的补偿性操作进行细节的分析了。相反，我会将“既成事实”直接告诉大家，详情如下。

```
ON INSERT is INTO S , INSERT isp INTO SP :
   INSERT ( is JOIN SP ) INTO SSP ,
   INSERT ( S JOIN isp ) INTO SSP ;

ON DELETE ds FROM S , DELETE dsp FROM SP :
   DELETE ( SSP MATCHING ds ) FROM SSP ,
   DELETE ( SSP MATCHING dsp ) FROM SSP ;

ON INSERT i INTO SSP :
   INSERT i { SNO , PNO , QTY } INTO SP ,
   INSERT i { SNO , CITY } INTO S ;

ON DELETE d FROM SSP :
   DELETE ( SP MATCHING d ) FROM SP ,
   DELETE ( ( S MATCHING d ) NOT MATCHING SSP ) FROM S ;
```

下面让我们来看一些例子，它们都将套用图 8.1 的样本值。注意：现在我们的首要任务是研究关系变量 S 和 SP 是基础关系变量，而关系变量 SSP 是视图的情况，那么我将首先把精力集中在对关系变量 SSP 的更新上。

1．假设我们从关系变量 SSP 中删除数组（S1，London，P1，300），那么效果将是从关系变量 SP 中删除数组（S1，P1，300），而（因为在原始删除操作执行后，关系变量 SSP 中仍然保留着一些供应商 S1 的数组，所以）保持关系变量 S 不变。

2．假设我们从关系变量 SSP 中删除数组（S3，Paris，P2，400），那么效果将是从关系变量 SP 中删除数组（S3，P2，400），并且（因为在原始删除操作执行后，关系变量 SSP 中已经没有任何供应商 S1 的数组了，所以）也从关系变量 S 中删除数组（S3，Paris）。注意：在这里我要提醒大家，我们已经假设在本例中每一个供应商至少都供应一种零部件。

3．假设我们向关系变量 SSP 中插入数组（S2，Paris，P3，500），那么效果将是

[1] 在这里我要引用希斯定理，这在第 5 章中我们也曾提到过多次。请大家参看《Database Design and Relational Theory》来获得更进一步的解释。

向关系变量 SP 中插入数组（S2，P3，500），而保持关系变量 S 不变。

4．假设我们向关系变量 SSP 中插入数组（S5，Athens，P3，500），那么效果将是向关系变量 SP 中插入数组（S5，P3，500），并且也向关系变量 S 中插入（S5，Athens）。

至于显式 UPDATE 操作，我不再赘述细节了，凡是属于下面 3 种更新的需求都可以顺利执行：（a）要改变某些设备供应的供应商号码、零部件号码或者数量的，（b）要改变某些供应商的城市的，或者甚至（c）要改变供应商的供应商号码的。

接下来我们来看看对于关系变量 S 和 SP 的更新。

1．假设我们从关系变量 SP 中删除数组（S1，P1，300），那么效果将是从关系变量 SSP 中删除数组（S1，London，P1，300），并且（因为在原始删除操作执行后，关系变量 SSP 中仍然保留着一些供应商 S1 的数组，所以）保持关系变量 S 不变。

2．假设我们从关系变量 SP 中删除所有关于供应商 S1 的数组，那么效果将是从关系变量 SSP 中删除所有关于供应商 S1 的数组，并且从关系变量 S 中删除数组（S1，20）。

3．假设我们向关系变量 SP 中插入数组（S2，P3，500），那么效果将是向关系变量 SSP 中插入数组（S2，Paris，P3，500）。

4．假设我们尝试向关系变量 SP 中插入数组（S5，P3，500），那么如果没有同时向关系变量 S 进行相应插入操作的话，该插入将会失败。

5．假设我们尝试向关系变量 S 中插入数组（S5，Athens），那么如果没有同时向关系变量 SP 进行相应插入操作的话，该插入将会失败。

6．假设我们从关系变量 S 中删除数组（S1，20），那么效果将是从关系变量 SSP 中删除所有关于供应商 S1 的数组。当然，这时候对于在 SSP 上删除的规则将开始起效，这将会（并不合理地）导致系统从关系变量 SP 中删除所有关于供应商 S1 的数组。

至于显式 UPDATE，我将把下面两个项目的证明工作交给大家作为练习：（a）鉴于我们在第 4 章有过相应的讨论，可以想见如果我们想要改变某个供应商的供应商号码时，我们没有什么别的选择而只能自己写出必要的双 UPDATE 语句，如下面的例子。

```
UPDATE S  WHERE SNO = ‘S1’ : { SNO := ‘S9’ } ,
UPDATE SP WHERE SNO = ‘S1’ : { SNO := ‘S9’ } ;
```

（b）但是其他的所有显式 UPDATE 操作都可以正常执行。

最后，我要来总结一下，前面所讲的这些规则，尤其是对关系变量 SSP 的更新适

用的，从本质上都与在第 6 章和第 7 章中我所给出的连接更新规则一致。正因为如此，我不会再对下面两种情况进行详细说明：（a）用户只能够看到关系变量 SSP，（b）关系变量 SSP 是视图，而 S 和 SP 是基础关系变量。因为已经有足够的证据可以证明在这两种情况下所有的更新都可以正常执行。

## 8.2 示例 2：信息隐藏

下面我将要修改一下上面的例子，并且假设，与我们原始的供应商-零部件数据库中一样，并不是所有的供应商都必须至少供应一种零部件。在这种情况下，信息等价就消失了。具体来说就是由关系变量 S 和 SP 整合而成的这种设计可以表达并不供应任何零部件的供应商（就如我们选取供应商-零部件数据库常用样本值时的供应商 S5），而仅由连接 SSP 构成的设计显然无法表达这类信息。同样地，现在已经无法再确保在 SNO 和 CITY 这两个属性上，关系变量 S 与关系变量 S 和 SP 的连接 SSP 的投影是等价的了。而另一方面，现在至少我们还可以确保在 SNO、PNO 和 CITY 这 3 个属性上，关系变量 SP 与关系变量 S 和 SP 的连接 SSP 的投影依然是等价的。

尽管如此，下列几个约束还是适用于修改版的数据库。

- 往常的键约束全部起效：{SNO}是 S 的键，而{SNO,PNO}是 SP 和 SSP 各自的键。
- 在任意给定时间里，SSP 的当前值都等于 S 和 SP 当前值的连接。
- FD {SNO} → {CITY}在 SSP 中起效，当然在 S 中也同样起效。

通过扩展图 8.1 并恢复供应商 S5 的相关数组，我们就可以得到一个数据库值能够满足上述条件的实例，详情请参看图 8.2，并请特别注意观察关系变量 SP 与 SSP 在该图当前值下的关系，与它们在图 8.1 中的关系是相同的。

S

| SNO | CITY |
|---|---|
| S1 | London |
| S2 | Paris |
| S3 | Paris |
| S4 | London |
| S5 | Athens |

SP

| SNO | PNO | QTY |
|---|---|---|
| S1 | P1 | 300 |
| S1 | P2 | 200 |
| S1 | P3 | 400 |
| S1 | P4 | 200 |
| S1 | P5 | 100 |
| S1 | P6 | 100 |
| S2 | P1 | 300 |
| S2 | P2 | 400 |
| S3 | P2 | 200 |
| S4 | P2 | 200 |
| S4 | P4 | 300 |
| S4 | P5 | 400 |

SSP

| SNO | CITY | PNO | QTY |
|---|---|---|---|
| S1 | London | P1 | 300 |
| S1 | London | P2 | 200 |
| S1 | London | P3 | 400 |
| S1 | London | P4 | 200 |
| S1 | London | P5 | 100 |
| S1 | London | P6 | 100 |
| S2 | Paris | P1 | 300 |
| S2 | Paris | P2 | 400 |
| S3 | Paris | P2 | 200 |
| S4 | London | P2 | 200 |
| S4 | London | P4 | 300 |
| S4 | London | P5 | 400 |

图 8.2　图 8.1 的修改版

关于术语的注释：本章的标题展示了，或者说至少是强烈暗示了 S JOIN SP 依然是一个一对多连接，但是你必须理解这里所说的“多”是包含零的情况的。事实情况是，术语“一对多”在使用上经常不是很严格，包含以下两种情况。（a）所谓的严格符合的情况，比如我们在第 8.1 节讨论的示例 1；（b）像我们目前正在讨论的这个案例，它可能出现这种情况，即一方含有某元素，而另一方却没有与之相对应的元素。下面是一段轻微编辑过的文字引自《The Relational Database Dictionary, Extended Edition》（Apress 出版，2008 年）。

严格地讲，一个一对多的对应关系是对两个集合 s1 和 s2（不是必需不同）配对的规则，它规定 s1 中的每一个元素都至少对应一个 s2 中的元素，s2 中的每一个元素则都只能对应一个 s1 中的元素。然而，这个术语却经常被使用得不那么严格，从而对以下 3 种情况进行配对：（a）s1 中的每一个元素都对应任意数量的 s2 中的元素（数量可能是 0），而 s2 中的每一个元素都只对应一个 s1 中的元素；（b）s1 中的每一个元素都至少对应一个 s2 中的元素，而 s2 中的每一个元素都至多对应一个的 s1 中的元素；（c）s1 中的每一个元素都对应任意数量的 s2 中的元素（数量可能是 0），而 s2 中的每一个元素都对应至多一个的 s1 中的元素。除非想要表达的意思非常明确，否则我们最好避免使用这个术语。

虽然文中最后一句话是这么写的，但我在本章中还是偶尔会使用一下这个术语，不过我想我的指示意义在上下文之间都是很明确的。

我想向大家再次说明的是，正因为 S 和 SP 的连接 SSP 有信息丢失，所以我们无法再保证关系变量 S 与连接的投影在相应的属性上等价。更具体点来说就是任何可以由关系变量 SSP 单独表达的信息一定可以被关系变量 S 和 P 的整合所表达，而反之则不成立。（这里举一个后者中元素在前者中没有对应元素的查询例子，那就是“取得一个不供应零部件的供应商的供应商号码”。）。由此导致的结果就是显然这里会有特定的更新可以在 S 上执行，但却没办法相应地在 SSP 中进行。一个例子就是“向 S 插入一个关于供应商 S9 的数组”，而没有同时向 SP 中插入至少一个关于相同供应商 S9 的数组。

我并不想在这里对这个例子做过多的调研分析。相反，我要直接援引第 6 章中对示例 2 的分析，正如我在第 7 章中的做法一样。更具体地讲，就是我要说在第 6 章中对那个例子的细节分析稍做修改也完全可以适用于现在这个例子。换句话说，就是我认为在本章前面为严格的多对多连接视图更新所修改过的规则，如果需要可以直接应用于本例子中。改述一句我们在第 6 章讨论中的话，如果我们能够认可这个观点，那么我们至少能够获得一系列通用的一对多连接视图更新规则，这些规则总是有效并且在多对多连接应当严格符合条件时保证它们确实符合，更不用说这个规则

其实对所有连接视图都有效，不管连接是一对一、一对多，还是多对多了。除此之外，如果这些规则有时因为某种原因偶尔会出现不好用的情况，那么这里也有特定的解决办法，比如用数据库管理系统授权的子系统来禁止某些特定的更新，这可以解决以上问题。

## 8.3 小结

我们终于来到了这 3 章对连接更新讨论的结尾。不过在结束时，我还有一些观点要说（它们都或多或少地与我们的主题相关，不过你当然也可以忽略它们）。第一个问题是关于外键的。大家可能注意到了，在第 6、7、8 这 3 章中我并没有明显提及外键本身（其实在第 5 章对于投影部分也没有提）。实际上，这个省略是我有意为之的。而究其原因则是，在数据库界，有大量的人存在这样一种模糊的概念，那就是对连接进行更新与其“到底是根据外键定义还是根据相应的目标键定义的”有些关系，而我则有意地想要回避这个概念。话说回来，在我的例子中确实有一些是“外键连接”（如果非要用这种术语的话），但并不都是，尤其在第 7 章中使用的例子都不是。

不管怎样，我还是要针对外键在 Tutorial D 中的情况做一些说明。大家请看下面这个例子，我们在第 2 章中曾经讲过，它是我们常用来定义外键的普通语句。

```
VAR SP BASE RELATION
  { SNO CHAR , PNO CHAR , QTY INTEGER }
    KEY { SNO , PNO }
    FOREIGN KEY { SNO } REFERENCES S
    FOREIGN KEY { PNO } REFERENCES P ;
```

但是，我在第 2 章中也说过，像上面用到的 FOREIGN KEY 这种表述法本质上就是约束的简写方式，同样的定义完全可以用显式 CONSTRAINT 语句来表达，只是可能会写得更冗长一些。更具体点来说，我在第 3 章曾经展示过这样一个例子，{SNO}是关系变量 SP 的一个外键引用了关系变量 S 的键{SNO}，它可以通过关系运算表达为一个多元约束，详细如下。

```
FORALL x ∈ SP ( UNIQUE y ∈ S ( x.SNO = y .SNO ) )
```

（我提醒大家，UNIQUE 这个量词可以读作“这里存在唯一一个……”。）那么你可能会问 Tutorial D 的与这个关系运算表达式等价的“完整版”是什么样的。

下面就是我们用 Tutorial D 写出的语句。

```
CONSTRAINT ... IS_EMPTY ( SP NOT MATCHING S ) ;
```

但是这个声明并不能完全靠自己来实现我们想要的效果。它所说的是每一个设备供应都拥有“至少”一个相应的供应商。然而我们想要说的是每个设备供应只拥有“恰好”一个相应的供应商[2]。那么其实我们还可以用另一种方便的简化方式来达到目的，这种方式叫作“镜像关系”（关于这个概念详情请参阅《SQL and Relational Theory》）。具体来说，我们可以这样写：

```
CONSTRAINT ... IS_EMPTY ( SP WHERE COUNT ( ‼S ) ≠ 1 ) ;
```

解释说明：对于给定的 SP 数组，表达式‼S 读作“棒棒 S”或者“双棒 S”，表示 S 的数组集合具有和 SP 数组相同的供应商号码。因此，该约束整体要求每一个 SP 数组都只能拥有恰好一个相对应的 S 数组。

我们还是接着说 Tutorial D 语句，你可能还会感兴趣如何用来写出函数依赖（FD）的表达式。你可能已经注意到，我并没有给出任何与使用了 FD 的例子有关的表达式。实际上我们还是有几种方式来表达的，而其中最为简洁的也许就是我下面所列出的方式。

```
CONSTRAINT ... SSP { SNO , CITY } KEY { SNO } ;
```

你可以将这个CONSTRAINT声明读作：“如果我们要定义一个由SSP关于SNO和CITY的投影组成的关系变量，那么该关系变量应该拥有键{SNO}”，这个约束可以有效地达到我们的目的，因为它隐性地声明了下列FD在关系变量SSP中起效[3]。

```
FD { SNO } → { CITY }
```

大家可以回忆一下第 5 章的一个注脚，说的是如果 K 是关系变量 R 的键，那么 FD K → X 在 R 中存在，对所有具备 R 关系头的子集 X 起效。

作者注：请大家注意这一点，由上面的例子可以得出，我们应该有能力通过任意复杂度的关系表达式来清楚地表达键。后面的章节我还会在引用到这一点。

最后，你可能还有疑问，我们是否需要一些特别的语句来表示连接依赖（JD）或者多值依赖（MVD）。这个我想我们没有。例如，我们可以参考第 7 章的第一个例子，如果我们详细解读下面这个约束：

---

[2] 其实在某种情况下这个声明是可以达到目的的，那就是这里需要有另外一个约束使 S 中的供应商号码都是唯一不重复的。但是这样就必须从另外两个被隐性声明的约束中推断出这个还比较重要的约束隐性起效，这至少从用户的角度来看是有点不友好的。

[3] 其实它隐性地声明了 FD {SNO} → {CITY}在关系变量 SSP 的 SNO 和 CITY 上的“投影”中起效。但是我们很容易证明一个给定的 FD 当且仅当它在给定的关系变量 R 自身中起效的时候，它可以在该 R 的给定投影上起效。

```
CONSTRAINT ... SCP = JOIN { SCP { SNO , CITY } , SCP { PNO , CITY } } ;
```

会发现它等价于我们对下面这个在关系变量 SCP 中起效的 JD 所做的声明。

```
☼{ { SNO , CITY } , { PNO , CITY } }
```

它同样也等价于在关系变量 SCP 中起效的特定的一对 MVD 的声明。与 FD 的情况不同，我们说某个关系变量等价于其特定投影的连接，并“不”等于说明某些 FD 起效，而这也是为什么通常我们都要对 FD 进行单独的具体说明的原因。

# 第 9 章

# 交集视图

更新交集，有时需要修正。

——Anon：《Where Bugs Go》

在本章以及接下来的两章中，我将把我的重心放在交集视图、合并视图以及差异视图上。当然，我已经在第 6 章中讲了一些关于交集视图的内容（也就是本章的标题），那就是交集是一对一连接的特殊情况，不过关于这个议题我们还有很多需要讲的。

首先我要说一说本章的行文结构。令DB1 和DB2 分别为一个基础关系变量集合和一个视图集合。在前面几章中，我对限制、投影以及连接视图等主题大体做了如下套路的介绍：首先，我会分析一个具体的DB1；其次，我会分析一个与DB1 等价的具体的DB2；最后，我会分析另一个DB2，这个DB2 并不等价于DB1，而是引入了某些信息隐藏。但是，这种方式并不太适用于交集视图[1]。原因是这样的：假设DB1 包含基础关系变量A和B；而DB2 只包含视图V，V定义为A与B的交集，即V = A INTERSECT B。那么只有在A=B的情况下DB2 才能信息等价于DB1，但是A=B这种情况下（我在第 6 章临近末尾的一个注脚中有提示过）定义视图V就没有什么意义了。

也就是说，A 和 B 相等这种情形我们研究起来其实并没有太大意义。除此之外，研究 A 和 B 完全不相交的情况也没有什么意义，在这种情况下没有任何数组能够同时出现在两个关系变量中。如果它们完全不相交的话，那么交集视图 V 将永远

[1] 这种方式同样也不太适用于后面两章的主题——合并视图和差异视图，只是原因不同。

为空，因此在 V 上的删除操作总是“空操作”，而向 V 的插入操作也总会失败（除非插入的也是“空操作”）。所以我想要探讨的有意思的例子是 A 和 B 并不等价，但是至少是有重叠的，这样的话相同的数组就有可能同时分别出现在二者之中。因此，下面我们将要讨论两类主要的例子，一种情况是二者的重合是显式的（后面我会解释这个说法的意思），而另一种则是二者的重合仅仅是隐性的。另外，我不会再去验证这两类例子中所有关系变量都是基础关系变量的情况，相反我会给定两个关系变量 A 和 B 开始，并立刻定义它们的交集 A INTERSECT B 为一个视图，从这里开始对它们进行观察与分析。

## 9.1 示例 1：显式重叠

我的第 1 个例子是基于在第 4 章 4.9 节“重叠限制”中已经分析过的针对限制视图的例子。它涉及到两个关系变量 NLS（“非 London 供应商”）和 NPS（“非 Paris 供应商”），详情如下。

```
NLS { SNO , SNAME , STATUS , CITY } KEY { SNO }
NPS { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

当然，如果按照我们惯常所见的供应商-零部件数据库来看，这些关系变量大概都是视图，具体来讲都是关系变量S的视图，但是你也大可以把它们视为基础关系变量。除此之外，观察可知这些关系变量显然是有重叠的，更具体地说就是关于那些既不在London，又不在Paris的供应商的数组将会同时出现在这两个关系变量中。（确实，这类数组“必须”同时出现在二者之中，而这要归功于下面所列出的相应的谓词，以及“闭域假设”。）[2] 那么现在也许你可以看出为什么我会说在这个例子中重叠是显式的了。一个数组想要同时出现在两个关系变量中则必须满足以下条件：（a）可以通过验证明显存在于该数组中的值的方式来检查，（b）确实能够被显式地影射到相应的约束中（后面我会验证）。注意：正式地来讲，上面所说的条件是一个简单的限制条件（关于这个概念的定义可以参考附录B）。

下面所列的是各个关系变量的谓词。

NLS：供应商 SNO 已经签约，名字是 SNAME，拥有状态 STATUS，并且位于城市 CITY（CITY 不是 London）。

NPS：供应商 SNO 已经签约，名字是 SNAME，拥有状态 STATUS，并且位于城市

[2] 关于这点这里要说明一下，我们这里遇到了和第 4 章相同的局面，那就是谓词违反了原则，这个在第 2 章的注脚中谈到过，而这样导致的效果就是关系变量 R1 和 R2 的谓词最好应当是可以预防同一个数组能够同时满足它们两个的可能性的。

CITY（CITY 不是 Paris）。

下面我们定义关系变量 NLS 和 NPS 的交集为视图 XLP。

```
XLP { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

这个关系变量的谓词如下。

XLP：供应商 SNO 已经签约，名字是 SNAME，拥有状态 STATUS，并且位于城市 CITY（CITY 既不是 London，也不是 Paris）。

样本值如图 9.1 所示。

NLS

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S5 | Adams | 30 | Athens |

NPS

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

XLP

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S5 | Adams | 30 | Athens |

图 9.1　关系变量 NLS、NPS 以及 XLP 的样本值

在这个例子中有哪些约束起效呢？首先是我们已经说过的，{SNO}分别是 3 个关系变量的单键。其次，{SNO}在 XLP 中是外键，同时引用了 NLS 和 NPS。最后，我们显然还拥有以下约束。

```
CONSTRAINT ... XLP = NLS INTERSECT NPS ;
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
CONSTRAINT ... IS_EMPTY ( NPS WHERE CITY = 'Paris' ) ;
CONSTRAINT ... ( NLS WHERE CITY ≠ 'Paris' ) =
                           ( NPS WHERE CITY ≠ 'London' ) ;
```

请大家特别注意最后一条，它表明对 NLS 和 NPS 各自的特定更新必须相应地级联到对方之中。

```
ON INSERT i INTO NLS : INSERT ( i WHERE CITY ≠ 'Paris' ) INTO NPS ;
ON INSERT i INTO NPS : INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS ;

ON DELETE d FROM NLS : DELETE ( d WHERE CITY ≠ 'Paris' ) FROM NPS ;
ON DELETE d FROM NPS : DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ;
```

于是我们就有了下面几点需要注意。

- 首先，大家可能注意到了，上面这几条规则中并没有提到关系变量 XLP，实际上即使 XLP 不存在，它们也还是被需要的。

- 在删除规则中，其实如果我们删掉（或者说，直接用 TURE 代替）限制条件 CITY≠‘Paris’和 CITY≠‘London’也不会有多大不同。我把它们加进来是为了让大家看得更明确，同时也是为了与插入规则对称。

- 正如在第 4 章中提到过的，上面所列约束的最后一条，就是那条表明必须级联的约束，同时也表明关系变量 NLS 和 NPS 合在一起违反了“正交设计原则”。不过，这对于目前我们的分析目的来说并不重要。

- 最后，我们需要一个约束来达到这样的效果，如果某个供应商同时出现在 NLS 和 NPS 之中，那么分别代表该供应商的两个数组应该是完全相同的。下面这个约束声明了{SNO}是 NLS 和 NPS 合并的键，它就可以达到这个效果：

```
CONSTRAINT ... UNION { NLS , NPS } KEY { SNO } ;
```

当然，如果关系变量 NLS 和 NPS 实际上是供应商关系变量 S 的视图的话，那么这个约束就会自动地被强制执行。

现在让我们回到规则本身。从 NLS 和 NPS 到交集关系变量 XLP 的级联又是怎么样的呢？其实这些规则都来得很直接。

```
ON INSERT i INTO NLS : INSERT ( i WHERE CITY ≠ ‘Paris’ ) INTO XLP ;
ON INSERT i INTO NPS : INSERT ( i WHERE CITY ≠ ‘London’ ) INTO XLP ;

ON DELETE d FROM NLS : DELETE ( d WHERE CITY ≠ ‘Paris’ ) FROM XLP ;
ON DELETE d FROM NPS : DELETE ( d WHERE CITY ≠ ‘London’ ) FROM XLP ;
```

同样的，删除规则中的限制条件也可以被去掉而不会产生什么不同。

最终，下面列出的是对于交集关系变量本身的更新规则。

```
ON INSERT i INTO XLP :
   INSERT ( i WHERE CITY ≠ ‘London’ ) INTO NLS ,
   INSERT ( i WHERE CITY ≠ ‘Paris’ ) INTO NPS ;

ON DELETE d FROM XLP :
   DELETE ( d WHERE CITY ≠ ‘London’ ) FROM NLS ,
   DELETE ( d WHERE CITY ≠ ‘Paris’ ) FROM NPS ;
```

那么我们又有几点需要注意。

- 让我们先聚焦在删除规则上来。只要我们的例子还和 XLP 有关，那么删除操作级联到关系变量 NLS 和 NPS 之一就是很重要的事情。但是只要操作级联到其中一个，就会导致对另一个的额外级联，而这里也没有一个好的原因让系统偏爱其中一个，至少从实际语法上而言(当然，假设两个关系变量都是可见的)，也可以说它是一个不产生差异的差异，所以我设定将级联同时作用于这两个关系变量，这既是为了对称，也是为了让例子更明确。
- 实际上，对插入规则来说我也有类似的观点（通常来讲，向一个交集做插入操作需要同时级联到涉事双方，而在目前这个例子中，则是只要级联到其中一方就将导致对另一方的额外级联）。
- 同样的，这里的限制条件也可以被去掉而不造成什么损失，而且这次对插入规则和删除规则都适用。（到底是为什么呢？）

当然，正如我在本章介绍中所解释的那样，我们现在讨论的是一个交集的例子，因此我们目前遇到的情况就是信息等价在定义阶段就是不成立的。也就是说，任何可以通过关系变量 XLP 单独表达的信息一定都可以被关系变量 NLS 和 NPS 整合后表达，但反之却不然。（有一个查询的例子“找到 London 的供应商”，它刚好可以体现后者的信息在前者没有对应信息的情况。）这种情况的后果就是显然会有特定的更新可以在 NLS 和/或 NPS 中执行，但在 XLP 中却没有相应的更新。我们举一个这类更新的例子，那就是“向 NPS 插入一个供应商 S9 的数组并且没有同时向 NLS 插入相同的数组”，即告诉数据库要进行如下更新：有个位于 London 的“新”供应商 S9。但是，除了所有上面所说的之外，尤其是缺少信息等价这条，至少我们知道所有在交集视图 XLP 上的更新都能够令人满意地正常执行。而能够这样令人满意的原因正是本示例中的重叠是显式的。

下面让我们来快速思考一下，一个用户只能看到关系变量 XLP，也就是一个信息隐藏的情况会发生什么。这个用户会有以下体验。

a．会知道该关系变量的谓词。

b．将会知道供应商号码都是唯一的，而且 CITY 值永远不会是 London 或者 Paris。

c．将不会知道任何补偿性操作。

这类用户既不能向该关系变量插入数组，也不能在该关系变量中更新任何供应商号码，因为这些操作都有潜在风险会违反该用户所不知道的约束。

最后，我还要说一点，如果 NLS 和 NPS 自己是视图的话，即关系变量 S 的视图，

那么 XLP 还可以被直接定义为 S 的限制条件，于是有：

```
S WHERE CITY≠'London' AND CITY≠'Paris'
```

让我们姑且叫它限制条件 XLP'。那么显然，在 XLP'上的更新应该与在 XLP 上的更新拥有相同的效果，而它们也确实会这样。具体来说情况如下。

- INSERT：不论是要向 XLP，还是 XLP'插入数组，都必须保证该数组的 CITY 值不能是 London 或者 Paris。
- DELETE：不论是要从 XLP，还是 XLP'删除的数组，也都一定拥有既不是 London，也不是 Paris 的 CITY 值。

我还有不少关于关系变量更新的内容要讲，确切地说，是关于视图更新的，第 14 章我会呈现给大家，它们的定义形式不同，却都等价。

## 9.2 示例 2：隐式重叠

对于第 2 个例子，让我们从供应商转换为零部件。假设，在任意给定时间，某些零部件正在销售，某些零部件在库存中，有些两种情况都有。那么再假设我们将这种情况定义给两个关系变量，每个关系变量只有一个属性PNO，其中一个关系变量PL给出正在销售的零部件号码，而另一个关系变量PK则给出在库存中的零部件号码。那么当然，这两个关系变量的交集，我们叫它XLK，给出的就是既在出售，又有库存的零部件号码。然而关键是我们无法仅仅通过观察一个原始零部件关系变量P中给定的数组就判断这个零部件到底是应该通过哪个关系变量表达，是PL、PK、两者共同，或者两者都不。换句话说，重叠在这里是隐式的，而非显式的[3]。因此，谓词就变得非常简单。

PL：零部件 PNO 正在销售。

PK：零部件 PNO 在仓库中。

XLK：零部件 PNO 既在销售，也在仓库中有存货。

[3] 通过观察大家可以发现，关系变量 PL 和 PK 与第 9.1 节的关系变量 NLS 和 NPS 在某些方面很相似，比如它们都违反了我们提到过的原则，那就是相应的谓词应当避免一个数组同时满足两个关系变量的可能性。而另一方面，它们又与 NLS 和 NPS 有不同点，那就是它们没有违反“正交设计原则”。这是因为这里并没有一个约束规定如果一个特定的数组出现在其中一个关系变量中，那么它也必须在另一个中出现。

样本值如图 9.2 所示。请注意，参考我们平时 PNO 的值来看，我猜想在这个例子中，零部件 P4 既不在售，也没有库存。

PL

| PNO |
|---|
| P1 |
| P2 |
| P3 |
| P6 |

PK

| PNO |
|---|
| P2 |
| P5 |

XLK

| PNO |
|---|
| P2 |

图 9.2　关系变量 PL、PK 以及 XLK 的样本值

至于约束，{PNO}显然分别是 3 个关系变量的单键，而 XLK 等价于其他两个关系变量的交集。

```
CONSTRAINT ... XLK = PL INTERSECT PK ;
```

对 PL 和/或 PK 的更新无疑如下所示。

```
ON INSERT i INTO PL : INSERT ( i INTERSECT PK ) INTO XLK ;
ON INSERT i INTO PK : INSERT ( i INTERSECT PL ) INTO XLK ;

ON DELETE d FROM PL : DELETE d FROM XLK ;
ON DELETE d FROM PK : DELETE d FROM XLK ;
```

那么对 XLK 的更新又如何呢？我们说，插入规则是很明显的。

```
ON INSERT i INTO XLK :
   INSERT i INTO PL , INSERT i INTO PK ;
```

至于删除规则，显然它应该足以让对XLK的删除操作只级联到PL和PK其中之一（当然至少也要级联到其中一个上）。然而，这里并没有好的理由让系统选择其中一个，而不是另一个，实际上，因此我们在这里遇到的情况又是歧义问题，这个我在第 6 章中已经详细讨论过——至少是它的一种特殊情况，而当时的讨论经过相应的调整后在这里也同样适用[4]。因此我推荐将下面所列的内容作为一个合适的删除规则(注意观察，你会发现它与第 9.1 节示例 1 的删除规则具有相同的格式，而这并不是偶然的)。

```
ON DELETE d FROM XLK :
   DELETE d FROM PL , DELETE d FROM PK ;
```

当然，我们再一次面对信息等价消失的情况。也就是说，任何可通过关系变量 XLK

[4] 我要提醒大家，David McGoveran 有一个办法可以解决歧义问题，我在第 15 章中将会讨论。不过请一定注意，歧义问题并没有出现在显式重叠的示例 1 中。

单独表达的信息一定都可以被关系变量 PL 和 PK 整合后表达，但反之却不然。（有一个查询的例子“找到正在销售却没有库存的零部件的零部件号码”，它刚好可以体现后者的信息在前者没有对应信息的情况。）这种情况的后果就是显然会有特定的更新可以在 PL 和/或 PK 中执行，但在 XLK 中却没有相应的更新。我们举一个这类更新的例子，那就是“向 PL 插入一个新数组，并且没有同时向 PK 插入相同的数组”，即告诉数据库某种零部件正在销售，但是如果之前没有库存，那就依然没有。

我将把下面这种情况留给大家作为练习：在我们上面所讲的基础上，如果有一个用户只能看到关系变量 XLK 的话，会是怎样的情形。不过很遗憾地告诉大家，这里还有一个问题（我曾在第 6 章提到过这个问题，但是并没有详细展开）。假设我们开始的时候只有关系变量 PL 和 PK，没有交集视图，同样假设零部件 P2 同时在两个关系变量中出现。那么下面的更新显然是合法的。

```
DELETE ( P2 ) FROM PL ;
```

请尤其注意观察，这个删除操作是不会级联到关系变量PK的。但是如果我们现在假设引入交集视图XLK，那么按照上面给定的规则，这个删除操作将会级联到关系变量PK！更有甚者，即使视图XLK对发起这个在关系变量PL上的删除操作的用户并不可见，这个级联删除还是会发生。或者我们换个说法，引入这个视图就必然要同时引入PL和PK之间相互的级联删除规则 [5]。

这样的情况显然是不尽如人意的。我们能做点什么来改善它吗？有一个可能奏效的实际的改进方法，虽然有点烦人，不过对目前这个例子应该可行，那就是在删除操作之后立即执行一个插入操作，于是有：

```
DELETE ( P2 ) FROM PL ;
INSERT ( P2 ) INTO PK ;
```

不过就我个人而言，我认为接下来这一节所要讨论的方法才是一个更加可取的方案。

### 9.2.1 更好的设计

假设我们完全将关系变量 PL 和 PK 用一个单独的关系变量替换掉，让我们叫它 POI，带有属性 PNO、ON_SALE 以及 IN_STOCK，其中属性 ON_SALE 和 IN_ STOCK 的类型是 BOOLEAN（并且拥有明显的解释），而{PNO}则是单键。那么对这样一

[5] 有一个相关的问题如下：在当前声明的 XLK 更新规则下，从 XLK 中删除一个数组，然后再插入回来，可以保持状态不变，但是先向 XLK 插入一个数组然后再删除就不一定了。原因是向 XLK 插入数组可能导致一个数组被插入到 PL 和 PK 其中之一，而从 XLK 中删除数组则总是导致一个数组从两个关系变量中被同时删除。当然，关系变量 XLK 本身总是可以维持现状的。

个关系变量来说的一个可能的值如图 9.3 所示（请特别注意零部件 P4 的数组）。

POI

| PNO | ON_SALE | IN_STOCK |
| --- | --- | --- |
| P1 | TRUE | FALSE |
| P2 | TRUE | TRUE |
| P3 | TRUE | FALSE |
| P4 | FALSE | FALSE |
| P5 | FALSE | TRUE |
| P6 | TRUE | FALSE |

图 9.3　关系变量 POI 的样本值

现在我们定义关系变量POI的限制视图PL’和PK’，如下面的约束所示（我在这里使用PL’和PK’作为名字是为了避免与前面介绍的关系变量PL和PK发生混淆，不过当然关系变量PL’和PK’，也是为了与之前差不多的目的而设立的）[6]。

```
CONSTRAINT ... PL′  = POI WHERE ON_SALE ;
CONSTRAINT ... PK′  = POI WHERE IN_STOCK ;
```

接下来我们就可以定义 PL’和 PK’的交集 XLK’了，于是我们有：

```
CONSTRAINT ... XLK′  = PL′  INTERSECT PK′  ;
```

（当然，我们也可以直接将 XLK’定义为 POI 的直接限制条件，如 POI WHERE ON_SALE AND IN_STOCK。但是在这里让我们暂且忽略这种可能性直到有新的说明。）对照图 9.3 中所示的 POI 的样本值，下面给大家列出这几个新的限制条件的样本值，如图 9.4 所示。

PL′

| PNO | ON_SALE | IN_STOCK |
| --- | --- | --- |
| P1 | TRUE | FALSE |
| P2 | TRUE | TRUE |
| P3 | TRUE | FALSE |
| P6 | TRUE | FALSE |

XLK′

| PNO | ON_SALE | IN_STOCK |
| --- | --- | --- |
| P2 | TRUE | TRUE |

PK′

| PNO | ON_SALE | IN_STOCK |
| --- | --- | --- |
| P2 | TRUE | TRUE |
| P5 | FALSE | TRUE |

图 9.4　关系变量 PL’、PK’以及 XLK’的样本值

[6] 这里有一个问题留给大家：由关系变量 PL’和 PK’组成的设计，是否与由关系变量 PL 和 PK 所组成的设计信息等价？

各变量的谓词（故意展现了一些比较折磨人的细节）如下。

PL’：零部件 PNO，当且仅当 ON_SALE 为 TRUE 时在销售，当且仅当 IN_STOCK 为 TRUE 时有库存（而 ON_SALE 总为 TRUE）。

PK’：零部件 PNO，当且仅当 ON_SALE 为 TRUE 时在销售，当且仅当 IN_STOCK 为 TRUE 时有库存（而 IN_STOCK 总为 TRUE）。

XLK’：零部件 PNO，当且仅当 ON_SALE 为 TRUE 时在销售，当且仅当 IN_STOCK 为 TRUE 时有库存（而 ON_SALE 总为 TRUE 且 IN_STOCK 总为 TRUE）。

至于约束，注意首先这 3 个关系变量每一个都以{PNO}为单键，{PNO}为 XLK’的外键，同时引用 PL’和 PK’。下面所列的约束显然也起效。

```
CONSTRAINT ... IS_EMPTY ( PL′  WHERE NOT ( ON_SALE ) ) ;
CONSTRAINT ... IS_EMPTY ( PK′  WHERE NOT ( IN_STOCK ) ) ;
CONSTRAINT ... IS_EMPTY ( XLK′  WHERE NOT ( ON_SALE AND IN_STOCK ) ) ;
```

我们同样也需要一个约束达到这样的效果，如果某个零部件同时能够被 PL’和 PK’表达，那么这两个数组所代表的零部件实际上应该是同一个。

```
CONSTRAINT ... UNION { PL′ , PK′ } KEY { PNO } ;
```

当然，如果关系变量 PL’和 PK’像声明的那样确实是 POI 的视图，那么上述约束就会自动执行。

作者注：顺便说一句，我们同时还有下面这条约束。

```
CONSTRAINT ... ( PL′ WHERE IN_STOCK ) = ( PK′ WHERE ON_SALE ) ;
```

那么这条约束起效会给我们带来这样的结果，关系变量 PL’和 PK’，就像 9.1 节中的 NLS 和 NPS 一样，显然违反了“正交设计原则”。换句话说，这样似乎与我自己所推崇的设计有所违背！具体来说就是在《Database Design and Relational Theory》中，我曾大力推荐一定不能违反正交设计。不过当然这里的违反行为无伤大雅，因为由之而产生的冗余一定可以被很好地控制住。

除此之外，更新规则又是怎样的呢？我们首先必须考虑一下在关系变量 POI 本身的更新会如何影响关系变量 PL’和 PK’。

```
ON INSERT i INTO POI :
   INSERT ( i WHERE ON_SALE ) INTO PL′  ,
   INSERT ( i WHERE IN_STOCK ) INTO PK′  ;

ON DELETE d FROM POI :
```

```
DELETE ( d WHERE ON SALE ) FROM PL′ ,
DELETE ( d WHERE IN_STOCK ) FROM PK′ ;
```

接下来我们必须考虑的是在关系变量 PL’和 PK’上的更新又是如何影响关系变量 POI 的。

```
ON INSERT i INTO PL′ : INSERT i INTO POI ;
ON INSERT i INTO PK′ : INSERT i INTO POI ;
ON DELETE d FROM PL′ : DELETE d FROM POI ;
ON DELETE d FROM PK′ : DELETE d FROM POI ;
```

请注意上述规则合在一起会得出如下几条结果：（a）向 PL’或 PK’任一方中插入形如（p，TRUE，TRUE）的数组，都会级联到另一方当中；（b）类似的，从 PL’或 PK’任一方中删除形如（p，TRUE，TRUE）的数组，都会级联到另一方。实际上，可以确定的是这些级联是逻辑上的必然结果，即使 POI 以及将对它的删除和插入级联到关系变量 PL’和 PK’的规则不存在也是一样。

那么现在我们可以来说一说 XLK’了，换句话说，我们可以来考虑一下在 PL’和 PK’上的更新是如何影响关系变量 XLK’的，以及反过来的情形又如何，规则详见下面。

```
ON INSERT i INTO PL′ : INSERT ( i WHERE IN_STOCK ) INTO XLK′ ;
ON INSERT i INTO PK′ : INSERT ( i WHERE ON_SALE ) INTO XLK′ ;

ON DELETE d FROM PL′ : DELETE ( d WHERE IN_STOCK ) FROM XLK′ ;
ON DELETE d FROM PK′ : DELETE ( d WHERE ON_SALE ) FROM XLK′ ;

ON INSERT i INTO XLK′ :
   INSERT i INTO PL′ , INSERT i INTO PK′ ;

ON DELETE d FROM XLK′ :
   DELETE d FROM PL′ , DELETE d FROM PK′ ;
```

这些规则全部都是毫无争议的，我希望你也能同意[7]（虽然我想把它们与本节上一个版本的例子比较会是有益处的）。从效果上讲，我所做的其实是重新设计了数据库，把原来只能通过PL和PK隐式表达的信息变为现在可以由属性ON_SALE和IN_STOCK的值显式表达。而这种重新设计的效果就是将上一个版本的例子中挣扎于那些差强人意的更新行为的情况，转变为像本章示例 1 那种顺畅的情况。当然，后面这个例子也不必像上一个版本那样受到歧义问题和顺序更新问题的困扰。

作者注：能够给大家推荐一个通用的设计原则是一件很诱人的事情。具体如下：如果两个关系变量拥有同样的关系头，但是却拥有不同的语义，那么（a）介绍一个

[7] 在删除规则中，级联到关系变量 PL’和 PK’其中之一当然就足够了，不过从其中任何一个关系变量中进行删除无论怎样都会自动级联到另一个。

或几个最能够体现差异的属性，（b）考虑使用一个关系变量来替代几个目前的关系变量。还有一个是：如果某个关系变量 B 的值总是另一个关系变量 A 的值的子集，但是 B 又无法被定义为 A 的限制条件，那么就引入一个或几个 A 和 B 可以共用的属性来将 B 直接变为一个限制条件。不过就这个问题再多做讨论的话就超出本书的范围了。

我将把下面这种情况留给大家作为练习：在我们上面所讲的基础上，如果有一个用户只能看到关系变量 XLK'的话，会是怎样的情形。我要再给大家留个练习，如果 XLK'被直接定义为 POI 的限制条件，那么看看作为一个关于更新的限制条件是否与上面讲的作为交集的版本在本质上是一致的。

### 9.2.2 另一个可能的设计

现在你可能会觉得，既然在 PL'中属性 ON_SALE 的值在每个数组中都必须是 TURE，而在 PK'中属性 IN_STOCK 的值在每个数组中都必须是 TURE，那么我们其实可以重新定义这两个关系变量，或者更具体地说是这两个视图，将这几个属性从其中去掉，详情如下列约束所示。

```
CONSTRAINT ... PL′ ′ = ( POI WHERE ON_SALE ) { PNO , IN_STOCK } ;
CONSTRAINT ... PK′ ′ = ( POI WHERE IN_STOCK ) { PNO , ON_SALE } ;
```

（我在这里使用双引号来将这个版本的关系变量与之前的单引号版本区分开来。）下面是各变量的谓词。

PL''：零部件 PNO 正在销售，并且，当且仅当 IN_STOCK 为 TRUE 时，同时有库存。

PK''：零部件 PNO 有库存，并且，当且仅当 ON_SALE 为 TRUE 时，同时正在销售。

那么现在，我们可以将 XLK'重新定义为 XLK''，让它成为 PL''和 PK''在 PNO 的投影的交集。

```
CONSTRAINT ... XLK′ ′ = PL′ ′ { PNO } INTERSECT PK′ ′ { PNO } ;
```

而其谓词如同我们原始的 XLK 一样。

XLK''：零部件 PNO 正在销售并且有库存。

图 9.5 所示的是这个重新设计的数据库的样本值。你可能会发现它与图 9.2 对比起来很有意思。

现在，让我们先把精力集中在关系变量PL''上。当然这个关系变量是一个投影关系变量。更详细地说，它是关系变量POI的一个投影，同时也是它的一个特定限制条件。因此，如同在第 5 章中解释过的，在这里信息等价就不能成立，由此导致的结果

就是关系变量PL’’仅支持删除操作，但不支持插入操作[8]。相应的删除规则如下所示。

```
ON DELETE d FROM PL′ ′  : DELETE ( POI MATCHING d ) FROM POI ;
```

PL''

| PNO | IN_STOCK |
|---|---|
| P1 | FALSE |
| P2 | TRUE |
| P3 | FALSE |
| P6 | FALSE |

PK''

| PNO | ON_SALE |
|---|---|
| P2 | TRUE |
| P5 | FALSE |

XLK''

| PNO |
|---|
| P2 |

图 9.5　关系变量 PL’’、PK’’以及 XLK’’的样本值

类似的，我们也有下面的规则。

```
ON DELETE d FROM PK′ ′  : DELETE ( POI MATCHING d ) FROM POI ;
```

那么对 XLK’’的删除规则如下所示（但是请注意，这里没有任何插入规则存在）。

```
ON DELETE d FROM XLK′ ′  :
   DELETE d FROM PL′ ′  , DELETE d FROM PK′ ′  ;
```

换句话说就是删除操作可以在 XLK’’上正常执行（或许在这里会有点争议），而插入操作则完全无法在 XLK’’上执行。所以总地来说，我认为这个“双引号”设计基本就是一个可以放弃的想法，因为原本我们期望它至少也应该是一个能支持更新的设计，然而并不是。

## 9.3　小结

到现在，你可能会觉得这一章有点令人困惑，特别是在 9.2 节中的示例 2，我不停地改变游戏规则，我们姑且这样比喻。那么现在让我抽离那一节的例子，并总结一下当我们遇到交集时的更新规则。定义 V 为 A INTERSECT B。接下来，为了简便，假设要向 A 和 B 插入的数组必须分别满足布尔表达式 ax 和 bx，在这里 ax 和 bx 分别表示相应关系变量的限制条件（默认就是 TURE）。于是我们有：

```
ON INSERT i INTO A : INSERT ( i WHERE bx ) INTO V ;
ON INSERT i INTO B : INSERT ( i WHERE ax ) INTO V ;

ON DELETE d FROM A : DELETE ( d WHERE bx ) FROM V ;
```

[8] 如果规则允许，假设我们向 PL’’插入数组(p, k)，将会导致数组(p, TRUE, k)被插入到 POI 当中。问题在于，我们知道这件事，而数据库管理系统并不知道。注意：实际上，这个例子本质上等价于，或者至少说同构于我们在之前章节讨论过的例子，也就是那个我们曾想将属性 CITY 从关系变量 LS（“London Suppliers”）中去掉的例子。请大家参看第 5 章以获取更多信息。

```
ON DELETE d FROM B : DELETE ( d WHERE ax ) FROM V ;

ON INSERT i INTO V : INSERT i INTO A , INSERT i INTO B ;

ON DELETE d FROM V : DELETE d FROM A , DELETE d FROM B ;
```

我要给大家留个练习来验证，本章之前的例子中的所有给定的具体规则都是上述通用规则的特殊情况，有些可能是稍微“优化过”的特例。

同时我们也看到交集的删除规则有的时候会导致一些无法接受的结果，或者说至少在实际中很不让人满意的结果。尤其是我曾在第 6 章中指出，由于这些规则的效果相当于将逻辑 OR 视为逻辑 AND 或者反过来的情形，从而导致这种“令人不满”的情况出现在第 1 版的“零部件在销售”对“零部件有库存”的例子中。但是让我们再多思考一下这个例子……这个例子中真正的问题是给定一个具体的要从交集 XLK 中删除的数组，数据库管理系统无法判断该数组逻辑上应当依然属于关系变量 PL、关系变量 PK 或者两者都不。实际上，在那个例子中，对 PL 和 PK 相应的限制条件都简单地是 TURE（对比本章中之前讨论的 NLS 对 NPS 的示例 1 的情况）。而我所推荐的解决方案，实际上是重新设计数据库，从而让数据库管理系统可以判断给定的数组属于哪个（些）关系变量。我冒险推荐的这个方案在实践中总是起效，而且在除了视图更新以外的其他领域中可能也会有改善作用。

最后一点关于上面的总结：很显然的是我们在这个例子中遇到的情况又是信息被隐藏了。但是这次我们所说的信息隐藏与本书中之前出现的信息隐藏是有区别的。在早先的例子中，信息是“对某些用户”隐藏，因为如我之前所说这些用户只能看到全局的一部分。例如，第 1 章的例子中，用户可以看到限制 LS（“London suppliers”），却看不到限制 NLS（“non London suppliers”）。但是本章中的例子我们所遇到的情况是信息并没有对用户隐藏，而是对“数据库管理系统”隐藏。具体来说就是这次被隐藏的信息是如何判断一个给定的数组逻辑上到底属于 PL、PK 或者二者都不的标准。而如同已经说明的，我对于这个问题的解决方案就是改变数据库设计，从而让被隐藏的信息最终变得对数据库管理系统可见。

# 第 10 章

# 合并视图

告诉你一个好消息！更新合并视图很简单！

——Anon：《Where Bugs Go》

现在让我们来看看合并。你不会感到意外，本章的结构与第 9 章很相似。不过有一点需要提前介绍一下：A UNION B 与 A INTERSECT B 有一点很相似，那就是如果 A=B，那么结果将会很无聊。不过，与 A INTERSECT B 不同的是如果 A 与 B 交集为空的话，那就会相当有意思，至少是有那么点意思。所以我的第 1 个例子就是这样的一个，引入一个交集为空的合并。

## 10.1 示例 1：交集为空的合并

这个例子基本上就是第 1 章和第 4 章所讲的“抛砖引玉”的例子反过来。因此，我们有两个关系变量，LS(“London suppliers”)和 NLS(“non London suppliers”)，形式如下。

```
LS  { SNO , SNAME , STATUS , CITY } KEY { SNO }
NLS { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

当然，如果是在我们惯常所用的供应商-零部件数据库中，这些关系变量大概都是视图，是关系变量 S 的视图，但是你也可以把它们看作基础关系变量。除此之外，仔细观察这些关系变量会发现它们交集为空，因此同一个数组不可能同时出现在二者之中。下面是各变量的谓词。

LS：供应商 SNO 已经签约，名字为 SNAME，拥有状态 STATUS，并且位于城市

CITY（且 CITY 是 London）。

NLS：供应商 SNO 已经签约，名字为 SNAME，拥有状态 STATUS，并且位于城市 CITY（且 CITY 不是 London）。

下面让我们来定义 LS 和 NLS 的（交集为空的）合并视图 ULN。

```
ULN { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

该关系变量的谓词如下。

ULN：供应商 SNO 已经签约，名字为 SNAME，拥有状态 STATUS，并且位于城市 CITY（且 CITY 是 London 或者不是 London）。

（当然，如果 LS 和 NLS 确实是关系变量 S 的视图，那么 ULN 就与关系变量 S 完全相同。）样本值如图 10.1 所示。

LS

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S4 | Clark | 20 | London |

NLS

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S5 | Adams | 30 | Athens |

ULN

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

图 10.1　关系变量 LS、NLS 以及 ULN 的样本值

下面所列的约束，就是第 4 章相应的例子中的约束的修改版，显然适用于本示例，详细如下。

```
CONSTRAINT ... ULN = LS UNION NLS ;
CONSTRAINT ... DISJOINT { LS { SNO } , NLS { SNO } } ;
CONSTRAINT ... IS_EMPTY ( LS WHERE CITY ≠ ‘London’ ) ;
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = ‘London’ ) ;
CONSTRAINT ... LS  = ( ULN WHERE CITY = ‘London’ ) ;
CONSTRAINT ... NLS = ( ULN WHERE CITY ≠ ‘London’ ) ;
```

同时，如我们已经说过的一样，{SNO}是这 3 个关系变量的单键，且{SNO}在 LS 和 NLS 中是外键，引用 ULN。请注意，对比我们在第 9 章讲的交集的例子，在这里我们拥有信息等价，只由 ULN 构成的设计信息等价于由 LS 和 NLS 整合而组成的设计。

下面的补偿性操作同样是将第 4 章的版本稍做修改后拿来使用的，详细如下。

```
ON DELETE d FROM LS : DELETE d FROM ULN ;
ON DELETE d FROM NLS : DELETE d FROM ULN ;

ON INSERT i INTO LS : INSERT i INTO ULN ;
ON INSERT i INTO NLS : INSERT i INTO ULN ;
ON DELETE d FROM ULN : DELETE ( d WHERE CITY = ‘London’ ) FROM LS ,
                       DELETE ( d WHERE CITY ≠ ‘London’ ) FROM NLS ;

ON INSERT i INTO ULN : INSERT ( i WHERE CITY = ‘London’ ) INTO LS ,
                       INSERT ( i WHERE CITY ≠ ‘London’ ) INTO NLS ;
```

请注意，还是对比我们在第 9 章所讲的交集的例子，这里不存在对 LS 或 NLS 进行插入或删除操作之后必须要级联到另一方的问题。

对于只能看到关系变量 ULN 的用户来说，虽然没有深入讲解，但我也希望大家能够很容易地看出这个用户可以对其进行任何操作，就好像它真地是个基础关系变量一样。

下面我要说，如果 LS 和 NLS 确实是关系变量 S 的视图，那么 ULN 也可以直接被定义为 S 的限制，于是有：

```
S WHERE CITY = ‘London’ OR CITY ≠ ‘London’
```

让我们称这个限制为 ULN’。很显然，对 ULN’的更新应当与对 ULN 的更新效果相同，而它们也确实相同。详细来说如下。

- INSERT：要向不论 ULN 或者 ULN’中的哪一个插入数组，那么该数组拥有的 CITY 值必须是 London 或者不是 London，同时也必须被插入到另一个关系变量中。

- DELETE：要从不论 ULN 或者 ULN’中的哪一个删除数组，那么该数组拥有的 CITY 值一定是 London 或者不是 London，同时也必须被插入到另一个关系变量中。

（正如我在第 9 章中跟大家说过的，在第 14 章会跟大家讨论“定义不同却相互等价”的关系变量的更新问题。）

如你所见，这第 1 个合并的例子基本与第 1 章和第 4 章的“抛砖引玉”例子一样，因此关于它确实没有什么更多可说的。所以让我们赶快进入下一个例子吧，这个例子引入了一个有重叠的合并，而不是交集为空的。

## 10.2 示例 2：显式重叠

我的第 2 个例子与第 9 章的第 1 个例子很像，是基于第 4 章的 4.9 节讨论过的例子变化而来的。它引入了两个关系变量，NLS（“non London suppliers”）和 NPS（“non Paris suppliers”），分别如下。

```
NLS { SNO , SNAME , STATUS , CITY } KEY { SNO }
NPS { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

正如在第 9 章所说的，这两个关系变量确实有重叠，因此同一个数组有可能同时出现在二者之中。除此之外，它们的重叠是显式的，这个我们在上一章也已经解释过。下面我们再来列出它们的谓词。

NLS：供应商 SNO 已经签约，名字为 SNAME，拥有状态 STATUS，并且位于城市 CITY（且 CITY 不是 London）。

NPS：供应商 SNO 已经签约，名字为 SNAME，拥有状态 STATUS，并且位于城市 CITY（且 CITY 不是 Paris）。

现在让我们定义视图 ULP 为这两个关系变量的合并。

```
ULP { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

该关系变量的谓词如下。

ULN：供应商 SNO 已经签约，名字为 SNAME，拥有状态 STATUS，并且位于城市 CITY（且 CITY 既不是 London，也不是 Paris）。

（当然，如果 NLS 和 NPS 确实是关系变量 S 的视图，那么 ULP 就与关系变量 S 完全相同，与 10.1 节中 ULN 的情况相似。）样本值如图 10.2 所示。

NLS

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S5 | Adams | 30 | Athens |

NPS

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

ULP

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

图 10.2　关系变量 NLS、NPS 以及 ULP 的样本值

下列约束显然起效。

```
CONSTRAINT ... ULP = NLS UNION NPS ;
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
CONSTRAINT ... IS_EMPTY ( NPS WHERE CITY = 'Paris' ) ;
CONSTRAINT ... ( NLS WHERE CITY ≠ 'Paris' ) =
                         ( NPS WHERE CITY ≠ 'London' ) ;
```

同时，如我们已经说过的一样，{SNO}是这 3 个关系变量的单键，且{SNO}在 LS 和 NLS 中是外键，引用 ULP。请注意，在这里我们也是拥有信息等价的，只由 ULP 构成的设计信息等价于由 NLS 和 NPS 整合而组成的设计。注意：{SNO}作为 ULP 的键可以保证如果某个供应商同时出现在 NLS 和 NPS 中的时候，那么代表该供应商的两个数组其实就是同一个。当然，如果 NLS 和 NPS 是供应商关系变量 S 的视图的话，那么这个约束会自动生效。

下面是补偿性操作。首先，是从 NLS 向 NPS 级联以及其反向级联的更新规则（与第 9 章相应部分一致），详细如下。

```
ON INSERT i INTO NLS : INSERT ( i WHERE CITY ≠ 'Paris' ) INTO NPS ;
ON INSERT i INTO NPS : INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS ;

ON DELETE d FROM NLS : DELETE ( d WHERE CITY ≠ 'Paris' ) FROM NPS ;
ON DELETE d FROM NPS : DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ;
```

下面所列的则是在 NLS 和 NPS 上的更新级联到合并视图 ULP 的规则。

```
ON INSERT i INTO NLS : INSERT i INTO ULP ;
ON INSERT i INTO NPS : INSERT i INTO ULP ;

ON DELETE d FROM NLS : DELETE ( d WHERE CITY ≠ 'Paris' ) FROM ULP ;
ON DELETE d FROM NPS : DELETE ( d WHERE CITY ≠ 'London' ) FROM ULP ;
```

最后，下面所列的是在合并关系变量上的更新规则。

```
ON DELETE d FROM ULP :
   DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ,
   DELETE ( d WHERE CITY ≠ 'Paris' ) FROM NPS ;

ON INSERT i INTO ULP :
   INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS ,
   INSERT ( i WHERE CITY ≠ 'Paris' ) INTO NPS ;
```

下面是几个重点需要注意的地方。

- 让我们首先把注意力放到插入规则上。只要关系变量 ULP 被引入，那么对它进行的插入操作当然可以只级联到关系变量 NLS 和 NPS 其中之一。但是级联

到 NLS 或者 NPS 之中任何一个（只要逻辑上必要）都会导致一个额外的向另一方的级联操作，那么既然系统无从判断这两个关系变量哪个更好，至少从实际的语法上无法判断（当然前提是这两个关系变量都是可见的），我们可以说它们之间存在着一个无差别的差异，于是就规定级联是同时向两个关系变量发起的，这样一来也满足了对称和明确的要求。

- 实际上，删除规则也是类似的情况。通常来讲，从合并上删除数组是需要同时级联到两个有关的关系变量的。然而本示例中，如果逻辑上需要的话，级联到其中任何一个都会导致额外级联到剩下的一个。
- 对于删除规则来说，我们可以去掉限制条件而不会引起什么重大损失。

对于只能看到关系变量 ULP 的用户来说，我希望大家很容易能看出该用户可以正常执行任何操作就如同该关系变量是一个基础关系变量一样。

现在我要来说明，如果 NLS 和 NPS 确实是关系变量 S 的视图的话，那么 ULP 也可以被直接定义为 S 的限制，于是有：

```
S WHERE CITY ≠ ‘London’ OR CITY ≠ ‘Paris’
```

让我们姑且称之为 ULP’。那么显然，对于 ULP’的更新应当具有与对 ULP 的更新相同的效果，它们实际上也确实是这样。具体来说有：

- INSERT：被插入 ULP 或者 ULP’的数组所拥有的 CITY 值必须不是 London 或者 Paris，同时也必须被插入到另一个关系变量中。
- DELETE：被从 ULP 或者 ULP’中删除的数组拥有的 CITY 值一定不是 London 或者 Paris，同时也必须从另一个关系变量中被删除。

（再次声明，我在第 14 章会跟大家讨论许多在定义不同却相互等价的关系变量上的更新问题。）

正如你所见，本章的第 2 个示例与 10.1 节中讨论的例子并没有特别本质的区别（究其根本是因为它们都保留有信息等价）。那么接下来，让我们来看看当信息等价丢失的时候会是什么情况。

## 10.3 示例 3：隐式重叠

本节中的例子是将第 9 章中的示例 2 进行一定调整之后得来的。那么提醒大家一下，基本情况如下：在任意给定时间，某些零部件正在销售，某些零部件在仓库中，还

有一些两种状态都具备。此外，这种情况会通过两个关系变量表达，其中每一个都只带有单一的属性 PNO——其中一个关系变量（PL）提供正在销售的零部件号码，而另一个（PK）则提供在仓库中的零部件号码。这两个关系变量的合并，我们叫它 ULK，能够提供所有在销售的、在仓库的或者同时具备两种状态的零部件号码。然而重点是仅通过查看来源于原始零部件关系变量 P 的给定数组，我们无法判断该零部件应当属于 PL、PK、二者皆有或二者皆无。那么各关系变量的谓词如下所示。

PL：零部件 PNO 正在销售。

PK：零部件 PNO 拥有库存。

ULK：零部件 PNO 既在销售中，又拥有库存。

样本值如图 10.3 所示。

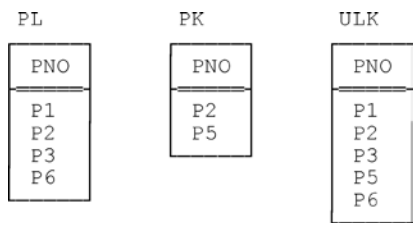



图 10.3　关系变量 PLPK 以及 ULK 的样本值

至于约束，{PNO}显然是这 3 个关系变量各自的单键，另外 ULK 等价于 PL 和 PK 的合并。

```
CONSTRAINT ... ULK = PL UNION PK ;
```

在 PL 和/或 PK 上的更新毫无疑问如下所示。

```
ON INSERT i INTO PL : INSERT i INTO ULK ;
ON INSERT i INTO PK : INSERT i INTO ULK ;

ON DELETE d FROM PL : DELETE ( d MINUS PK ) FROM ULK ;
ON DELETE d FROM PK : DELETE ( d MINUS PL ) FROM ULK ;
```

那么在 ULK 上的更新又是怎样的呢？首先，删除规则很明确如下所示。

```
ON DELETE d FROM ULK :
   DELETE d FROM PL , DELETE d FROM PK ;
```

至于插入规则，显然它应该足以将对 XLK 的删除操作只级联到 PL 和 PK 其中之一（当然至少也要级联到其中一个上）。然而，这里并没有好的理由让系统选择其中一

个，而不是另一个，实际上，我们在这里遇到的情况又是歧义问题，只是外在表象不太一样，我在本章最后一节对这个问题还会有更多讨论。现在，我推荐将下面所列的内容作为一个合适的插入规则（注意观察，你会发现它与第 10.1 节示例 1 和 10.2 节示例 2 的插入规则具有相同的格式，而这并不是偶然的）。

```
ON INSERT i INTO ULK :
   INSERT i INTO PL , INSERT i INTO PK ;
```

当然，如我早前所说过的，我们在这里又一次遇到了信息等价丢失的情况。具体一点说就是任何可通过关系变量 ULK 单独表达的信息一定都可以被关系变量 PL 和 PK 整合后表达，但反之却不然。（有一个查询的例子“找到正在销售却没有库存的零部件号码”，它刚好可以体现后者的信息在前者没有对应信息的情况。）这种情况的后果就是显然会有特定的更新可以在 PL 和/或 PK 中执行，但在 ULK 中却没有相应的更新。我们举一个这类更新的例子，那就是“从 PL 删除一个已经存在的数组并且没有同时从 PK 中删除相同的数组”，即告诉数据库某种零部件不再进行销售，但是如果之前有库存那就依然有。

我将把当用户只能看到关系变量 ULK 时对上述所有内容的验证留给大家作为练习。然而，很不幸的是这里还有另一个问题（在第 9 章隐式重叠的交集例子中也有讲过）。假设我们只从 PL 和 PK 这两个关系变量开始，没有合并视图。同时为了明确变量关系，也假设零部件 P4 既不在 PL 中，也不再 PK 中。那么下面的 INSERT 操作显然是合法的。

```
INSERT ( P4 ) INTO PL ;
```

请特别注意观察，这个插入操作没有级联到关系变量PK的可能性。但是如果现在假设我们引入合并视图ULK，那么，根据上面对规则的定义，这个插入操作就将会级联到关系变量PK！除此之外，即使视图ULK对发起这个插入操作的用户并不可见，系统也同样会执行级联。或者我们换一种说法：引入这个视图显然必须同时引入PL和PK之间相互的插入操作级联规则。[1]

现在，针对这个问题，尤其是目前我们手中的例子，有一种解决办法可能有效，那就是在插入操作之后立即执行一个相应的删除操作，具体如下。

```
INSERT ( P4 ) INTO PL ;
```

[1] 有一个相关的问题如下。根据对 ULK 更新规则的定义，向 ULK 中插入一个数组，然后立刻删除它，可以保持状态不变，但是先从 ULK 中删除一个数组然后立刻再插入回来则不一定能够维持状态不变。其原因在于从 ULK 中删除一个数组可能实际上会导致从 PL 和 PK 二者之一中删除一个数组，而向 ULK 插入一个数组则总是会导致一个数组同时插入到 PL 和 PK 中。当然，仅仅对关系变量 ULK 本身来讲，这两种操作都能够保证状态不变。

```
DELETE ( P4 ) FROM PK ;
```

不过，我其实更推荐在接下来的 10.3.1 节当中马上就会讨论到的解决方案（当然，它与第 9 章中讨论的解决交集视图中这个问题的方法从本质上讲是一样的）。

### 10.3.1 一个更好的设计

我在这里要展示给大家的解决方案可能会让大家回想起第 9 章中所介绍的方案，它将把关系变量 PL 和 PK 整体替换为一个单独的关系变量 POI，带有属性 PNO、ON_SALE 以及 IN_STOCK（其中属性 ON_SALE 和 IN_STOCK 的类型是 BOOLEAN 并且拥有明显的解释说明，{PNO}则是单键）。那么这个关系变量的一个可能的值如图 10.4 所示（与第 9 章的图 9.3 所示是一致的）。

POI

| PNO | ON_SALE | IN_STOCK |
|---|---|---|
| P1 | TRUE | FALSE |
| P2 | TRUE | TRUE |
| P3 | TRUE | FALSE |
| P4 | FALSE | FALSE |
| P5 | FALSE | TRUE |
| P6 | TRUE | FALSE |

图 10.4 关系变量 POI 的样本值

现在我们来定义限制视图 PL’和 PK’以及合并视图 ULK’，如下面的约束所示。

```
CONSTRAINT ... PL′  = POI WHERE ON_SALE ;
CONSTRAINT ... PK′  = POI WHERE IN_STOCK ;
CONSTRAINT ... ULK′  = PL′  UNION PK′  ;
```

图 10.5 展示的是这几个变量对应图 10.4 时的样本值。

PL’

| PNO | ON_SALE | IN_STOCK |
|---|---|---|
| P1 | TRUE | FALSE |
| P2 | TRUE | TRUE |
| P3 | TRUE | FALSE |
| P6 | TRUE | FALSE |

PK’

| PNO | ON_SALE | IN_STOCK |
|---|---|---|
| P2 | TRUE | TRUE |
| P5 | FALSE | TRUE |

ULK’

| PNO | ON_SALE | IN_STOCK |
|---|---|---|
| P1 | TRUE | FALSE |
| P2 | TRUE | TRUE |
| P3 | TRUE | FALSE |
| P5 | FALSE | TRUE |
| P6 | TRUE | FALSE |

图 10.5 关系变量 PL’、PK’以及 ULK’的样本值

它们的谓词如下所示。

PL’：零部件 PNO，当且仅当 ON_SALE 值为 TURE 时在销售，当且仅当 IN_STOCK 为 TURE 时有库存（而 ON_SALE 值总为 TURE）。

PK’：零部件 PNO，当且仅当 ON_SALE 值为 TURE 时在销售，当且仅当 IN_STOCK 为 TURE 时有库存（而 IN_STOCK 值总为 TURE）。

ULK’：零部件 PNO，当且仅当 ON_SALE 为 TRUE 时在销售，当且仅当 IN_STOCK 为 TRUE 时有库存（而 ON_SALE 总为 TRUE 或 IN_STOCK 总为 TRUE）。

至于约束，这 3 个关系变量每一个都以{PNO}为单键，而{PNO}为PL’和PK’的外键，引用ULK’。下面所列的约束也同时起效。[2]

```
CONSTRAINT ... IS_EMPTY ( PL′  WHERE NOT ( ON_SALE ) ) ;
CONSTRAINT ... IS_EMPTY ( PK′  WHERE NOT ( IN_STOCK ) ) ;
CONSTRAINT ... IS_EMPTY ( ULK′  WHERE NOT ( ON_SALE OR IN_STOCK ) ) ;
```

下面所列的是重复第 9 章的更新规则，规范了关系变量 POI 和关系变量 PL’和 PK’之间的联系。

```
ON INSERT i INTO POI :
   INSERT ( i WHERE ON_SALE ) INTO PL′  ,
   INSERT ( i WHERE IN_STOCK ) INTO PK′  ;

ON DELETE d FROM POI :
   DELETE ( d WHERE ON SALE ) FROM PL′  ,
   DELETE ( d WHERE IN_STOCK ) FROM PK′  ;

ON INSERT i INTO PL′  : INSERT i INTO POI ;

ON INSERT i INTO PK′  : INSERT i INTO POI ;

ON DELETE d FROM PL′  : DELETE d FROM POI ;

ON DELETE d FROM PK′  : DELETE d FROM POI ;
```

而下面所列的则是关系变量 PL’和 PK'与关系变量 ULK’之间的更新规则。

```
ON INSERT i INTO PL′  : INSERT i INTO ULK′  ;
ON INSERT i INTO PK′  : INSERT i INTO ULK′  ;

ON DELETE d FROM PL′  : DELETE ( d WHERE NOT ( IN_STOCK ) ) FROM ULK′  ;
```

[2] 在第 9 章中，我们还有一个规则规定限制 PL′WHERE IN_STOCK 和 PK′WHERE ON_SALE 必须是等价的，而这样会导致关系变量 PL’和 PK’违反“正交设计原则”。

```
ON DELETE d FROM PK′  : DELETE ( d WHERE NOT ( ON_SALE ) ) FROM ULK′  ;
ON INSERT i INTO ULK′  :
   INSERT ( i WHERE ON_SALE ) INTO PL′  ,
   INSERT ( i WHERE IN_STOCK ) INTO PK′  ;

ON DELETE d FROM ULK′  :
   DELETE d FROM PL′  , DELETE d FROM PK′  ;
```

我相信大家对上面这些规则也都没有异议（不过我认为把它们与上一个版本的例子做比较是很有意义的）。这里引用第 9 章的话：从效果上讲，我所做的其实是重新设计了数据库，把原来只能通过 PL 和 PK 隐式表达的信息变为现在可以由属性 ON_SALE 和 IN_STOCK 的值显式表达。而这种重新设计的效果就是让上一个版本的例子中挣扎于那些差强人意的更新行为的情况，转变为像本章示例 1 那种顺畅的情况。当然，后面这个例子也不必像上一个版本那样受到歧义问题和顺序更新问题的困扰。

我将把当用户只能看到关系变量 ULK’时对上述所有内容的验证留给大家作为练习。另外，我要再给大家留个练习，ULK’可以被直接定义为 POI 的限制条件，只要我们想，那么看看作为一个关于更新的限制条件是否与上面讲的作为合并的版本在本质上是一致的。最后，我还要给大家留一个练习，假设系统是可以支持相应的更新操作的，那么请大家证明这个例子的“双引号”版本是完全不可行的，这个版本类似于第 9 章示例 2 的那个“双引号”版本。

### 10.3.2 关于交集为空的合并

在这里我想简略地讨论一个有点不同的例子。假设某些零部件是在国外生产的，而另外的则是在国内生产的，二者没有重叠。继续假设我们将用两个关系变量来表达这种情况，二者都拥有单一的属性 PNO，其中一个关系变量（PA）表达在国外生产的零部件号码，另一个（PD）则表达在国内生产的零部件号码。请注意，这两个关系变量的合并，我们叫它 UAD，是一个交集为空的合并，并且能够直接提供所有的零部件号码。不过问题在于我们依然无法仅仅通过查看合并中的数组就判断出该数组到底是来自 PA，还是 PD（大家可以比对本章示例 1 的情况，那个例子中也引入了交集为空的合并）。各个关系变量的谓词如下。

PA：零部件 PNO 在国外生产。

PD：零部件 PNO 在国内生产。

UAD：零部件 PNO 是在国外生产的或者是在国内生产的（不可能既产自国外，又产自国内）。

那么现在，对比 10.3 节前面引入关系变量 PL 和 PK 的例子，即合并不是交集为空

而言，本例有一个重要的特点是下面这个约束显然是起效的。

```
CONSTRAINT ... DISJOINT { PA , PD } ;
```

这个约束将会导致如下结果：首先，通常来讲从 UAD 中进行删除操作将会成功，尽管删除任何给定数组都会级联到 PA 和 PD 其中之一；第二，可能是更重要的一点，那就是想 UAD 插入数组总会因为违反了黄金法则而失败（除非它们是“空操作”)。

## 10.4 小结

在结尾，让我从前面几节众多的示例讨论中抽离出来，并且给大家总结一下遇到合并介入时的更新规则。定义 V 为 A UNION B。接下来，为了更便于表达，我们假设所有要插入到 A 和 B 中的数组都必须分别满足布尔表达式 ax 和 bx，在这里 ax 和 bx 表示对相应关系变量的约束条件（默认为 TURE）。于是我们有：

```
ON INSERT i INTO A : INSERT i INTO V ;
ON INSERT i INTO B : INSERT i INTO V ;
ON DELETE d FROM A : DELETE ( d WHERE bx ) FROM V ;
ON DELETE d FROM B : DELETE ( d WHERE ax ) FROM V ;
ON INSERT i INTO V :
   INSERT ( i WHERE ax ) INTO A ,
   INSERT ( i WHERE bx ) INTO B ;
ON DELETE d FROM V :
   DELETE ( d WHERE ax ) FROM A ,
   DELETE ( d WHERE bx ) FROM B ;
```

本章前面例子的更新规则都是上述这些通用规则的特殊情况，有些可能是稍微“改进”的情况，对于这一点的验证，我将留给大家作为练习。

现在我们可以看到，在实际操作中对于合并的插入规则有时候可能会导致无法接受的结果，或者至少是不那么理想的结果。我就这个问题还要再详细说一说。令A和B的谓词分别为PA和PB，那么V = A UNION B的谓词就是PA OR PB。因此，如果我们向V中插入t，实际上就意味着命题PA(t) OR PB(t)为真。对比而言，同时向A和B插入t意味着PA(t) AND PB(t)为真。如果向V插入t导致t被同时插入A和B的话，那么实际上就又出现了逻辑OR与逻辑AND效果一样的情况 [3]。

我们还能看到，上述的结论适用于“零部件正在销售”对“零部件有库存”的第

[3] 在这里我提醒大家，David McGoveran 有一个办法来解决歧义问题，我将在第 15 章中进行讨论。

1 个版本的例子。然而，真正的问题在于给定一个特定的数组并将其插入到合并 ULK 中，数据库管理系统无法判断该数组逻辑上应当属于关系变量 PL 或关系变量 PK 或两者都有。实际上，在该例子中，对于 PL 和 PK 各自相应的限制条件就是简单的 TURE（与本章前面讨论过的示例 2 的情形，即 NLS 对 NPS 的例子形成对比）。而我对于这个问题的解决方案实际上就是改变数据库设计，从而让数据库管理系统最终可以判断给定的数组到底应该属于哪个（些）关系变量。我推荐的这个方案在实际操作中将会很有用，并且除了视图更新以外，在其他领域也能够提供很多便利。

# 第 11 章

# 差异视图

我来教教你什么叫差异：离开，离开！

——威廉·莎士比亚《李尔王》（1605）

在本章中，我将讨论通过关系差异运算符（在 Tutorial D 中是 MINUS）来进行更新的问题。本章的结构与前面两章——第 9 章和第 10 章的结构很相近，我相信你一定不会对这一点感到奇怪。但是，本章与前面两章相比，在某些重要的地方也有“差异”，这个词出现得多么切题啊！各位马上就能看到。让我来简单说明一下：假设我们给定两个关系变量 A 和 B。显然，如果 A=B，那么差异 A MINUS B 和差异 B MINUS A 的结果都会为空，所以这个例子并不怎么吸引人。同样地，如果 A 和 B 交集为空，那么 A MINUS B 等于 A，B MINUS A 等于 B，这个例子也没有什么意思。真正有意思的问题是 A 和 B 并不相等，但是却有重叠。因此，我将仿照第 9 章的方式，准备两个例子，其一是显式重叠，而另一个则是隐式重叠。但不同的是，这次我将先从隐式重叠开始说起，这样做的原因会在下面的内容中很快揭晓。

## 11.1 示例 1：隐式重叠

我的第 1 个例子还是基于第 9 章和第 10 章中“零部件正在销售”对“零部件有库存”的例子修改而来的。那么再一次地，我们拥有两个关系变量 PL 和 PK，它们每一个都只有一个属性 PNO——关系变量 PL 提供正在销售的零部件号码，而关系变量 PK 则给出有库存的零部件号码。我们显然至少有两个可能的差异可供考虑。为了一致，我们将讨论差异 PL MINUS PK（我们姑且叫它 DLK），它将给出我们那些正在销售却没有库存的零部件号码。那么，各个变量的谓词如下所示。

PL：零部件 PNO 正在销售。

PK：零部件 PNO 拥有库存。

DLK：零部件 PNO 正在出售并且没有库存。

样本值如图 11.1 所示。

PL

| PNO |
|---|
| P1 |
| P2 |
| P3 |
| P6 |

PK

| PNO |
|---|
| P2 |
| P5 |

DLK

| PNO |
|---|
| P1 |
| P3 |
| P6 |

图 11.1　关系变量 PL、PK 以及 DLK 的样本值

至于约束，{PNO}显然是 3 个关系变量各自的单键，当然还有 DLK 等价于另两个关系变量的差异，或者更确切地说，是两种可能的差异之一，具体如下。

```
CONSTRAINT ... DLK = PL MINUS PK ;
```

接下来看看补偿性操作，[1] 首先我们来看看对关系变量PL的删除规则。

```
ON DELETE d FROM PL : DELETE d FROM DLK ;
```

现在我们来想象一下从 PL 中删除数组 t。想要 t 出现在 DLK 中，条件就是只有当 t 出现在 PL 中，而不在 PK 中的时候才可以。而这个条件不管之前是否存在，在 PL 的删除操作之后肯定就不会存在了，因此才有了上面这条规则。

下面的是对关系变量 PL 的插入规则。

```
ON INSERT i INTO PL : INSERT ( i MINUS PK ) INTO DLK ;
```

现在我们再来想象一下向 PL 中插入数组 t。情况还是如此，那就是想要 t 出现在 DLK 中，条件是只有当 t 出现在 PL 中而不在 PK 中的时候才可以。而这个条件想要在对 PL 的插入操作后成立，就必须保证 t 在操作前不在 PK 中，并且操作之后依然不在 PK 中，因此才有了上面这条规则。

那么对于关系变量PK来说又是怎样的呢？首先我要指出，PL和PK在DLK的定义中处在非对称的角色上，因此按理说相应的补偿性操作也应当是非对称的，实际上它

[1] 我建议使用文氏图（Venn diagrams）来支持接下来的讨论。

们也确实如此。让我们先来讨论一下插入操作。再次想象一下插入数组t（这次是向PK插入）。现在t可能已经存在于PK中，也可能不存在。如果它已经存在于PK中，[2] 那么显然它在插入操作前不可能出现在DLK中，而操作之后依然不会出现。相反地，如果操作前它并不存在于PK中，那么即使开始时它出现在DLK中，插入操作后它也一定不会再出现。以上所有信息最终可以归纳为下面的规则。

```
ON INSERT i INTO PK : DELETE i FROM DLK ;
```

最后，让我们来想象一下从 PK 中删除数组 t。现在的情况是 t 存在于 PL 或者不存在于 PL 中。注意，这是“另一个”关系变量。如果 t 存在于 PL 中，那么无论它之前是否在 DLK 中，当删除操作结束后它都一定会在 DLK 中。反过来，如果 t 不存在于 PL 中，那么它显然不存在于 DLK 中，删除操作之后依然不在。上面的情况可以总结为下面的规则。

```
ON DELETE d FROM PK : INSERT ( d INTERSECT PL ) INTO DLK ;
```

在我们继续之前，我要说明一下，我们刚刚讨论的这两个补偿性操作是本书中目前为止第一个由插入操作导致删除或者反过来由删除导致插入的例子（换句话说就是，它们是本书第一个不是只通过简单的级联就能完成的补偿性操作例子）。

接下来我们继续说说 DLK 上的更新。下面首先是插入规则。

```
ON INSERT i INTO DLK :
   INSERT i INTO PL , DELETE i FROM PK ;
```

现在我们来想象一下向 DLK 中插入零部件号码 p 相关的数组。在这里零部件 p 正在销售，但没有库存。因此，如果 PL 中之前没有该数组，我们就必须向 PL 插入该数组，如果 PK 中之前有该数组，我们就必须从 PK 中删除该数组。请注意，整体上讲，这两个更新必须同时进行，去掉哪一个都是不完善的。如果我们只做插入，不做删除，那么最后效果就可能是零部件 p 既在销售，又有库存；类似地，如果我们只做删除，不做插入，那么最后的效果就可能是零部件 p 既不在销售，又没有库存。这两种情况都会导致零部件 p 的数组无法出现在 DLK 中，因此会造成违反对赋值原则的情况。

作者注：实际上对于上面的问题还有另一种思考方式（这种角度也许同时能够帮助你理解接下来就要讨论的删除规则）。如果用文氏图的话就很好理解了，任意给定差异 A MINUS B 可以被看作“交集”A INTERSECT C，这里 C 是 B 的（绝对）补

[2] 实际上我们可以忽略这种可能性，正如我们在第 4 章里面所看到的，没有任何规则会要求向变量中插入一个已经存在的数组，即使实际的表达式语句中有这样写也一样。

集。因此，对应第 9 章相应的讨论，一个对 A MINUS B 的插入操作应当会导致一个对 A 的插入操作和一个对 C 的插入操作。但是对 C 的插入操作在效果上其实就是对 B 的“删除操作”，它会导致数组被从 B 中拿掉并添加到 C 中去。

那么对于 DLK 的删除又是怎样的呢？我们有以下“对称”的规则。

```
ON DELETE d FROM DLK :
   DELETE d FROM PL , INSERT d INTO PK ;
```

这个规则怎么样？想象一下我们从DLK中删除零部件号码p的相关数组。在这里零部件p不是有库存，就是不在销售，还有可能既有库存，又不在销售。那么显然，无论是向PK插入数组，即零部件有库存，或者从PL中删除数组，即零部件不在销售，其中任意一种操作就足以达到我们想要的效果。实际上，如果像上面所说的两种操作都执行的话，就会再次出现逻辑OR等同于逻辑AND的效果。当然真正的问题在于这里并没有一个很好的理由从这两个操作中选一个，而不选另一个。也正因为如此，我还是推荐采用上面我所列出的规则，直到后面我另有说明为止。[3]

当然，我们在这里还有另一个问题那就是信息等价已经丢失。具体一点说就是任何可通过关系变量 DLK 单独表达的信息一定都可以被关系变量 PL 和 PK 整合后表达，但反之却不然。（有一个查询的例子“找到有库存却没有在销售的零部件的编号”，它刚好可以体现后者的信息在前者中没有对应信息的情况。）这种情况的后果就是显然会有特定的更新可以在 PL 和/或 PK 中执行，但在 DLK 中却没有相应的更新。我们举一个这类更新的例子，那就是“同时向 PL 和 PK 中插入某个新数组”，即告诉数据库某种零部件既在销售，又有库存。

我将把当用户只能看到关系变量 DLK 时对上述所有内容的验证留给大家作为练习。然而我必须指出，很不幸的是与第 9 章及第 10 章中讨论相似例子时一样，这里还有另一个问题。假设我们只从 PL 和 PK 这两个关系变量开始，没有差异视图。同时为了明确变量关系，假设零部件 P1 在 PL 中，但不在 PK 中，那么下列的操作显然是合法的。

```
DELETE ( P1 ) FROM PL ;

INSERT ( P1 ) INTO PK ;
```

请特别注意观察，上面的删除操作不会导致对关系变量PK进行插入操作，同样上面的插入操作也不会导致对关系变量PL发生删除操作。但是如果现在假设我们引

[3] 在这里我再次提醒大家，在第 15 章我将与大家讨论 David McGoveran 所推荐的一个方法来解决这类歧义问题。

入差异视图DLK。那么，根据上面对规则的定义，这个对PL的删除操作就将会导致对关系变量PK的插入操作，而这个对PK的插入操作也将会造成对PL的删除操作！除此之外，即使视图DLK对发起这两个插入操作的用户并不可见，系统也同样会这样执行。或者我们换一种说法，引入这个视图显然必须同时引入相应规则，以使得令对PL的删除操作引发对PK的插入操作，而令对PK的插入操作引发对PL的删除操作。[4]

现在对这个问题，特别是对我们手上的例子而言，有一个可能的解决办法，那就是在本来的删除或插入操作执行后立即执行另一个删除或者插入操作，详细如下。

```
DELETE ( P1 ) FROM PL ;
DELETE ( P1 ) FROM PK ;

INSERT ( P1 ) INTO PK ;
INSERT ( P1 ) INTO PL ;
```

不过，在接下来的小节中，我们马上就要讨论到的这个解决方案更值得推荐。当然，它本质上与第 9 章和第 10 章中讨论交集和合并视图时遇到相应问题的解决方案是一致的。

### 11.1.1　一个更好的设计

我在这里要展示给大家的解决方案可能会让大家回想起前面几章中所介绍的方案，它将把关系变量 PL 和 PK 整体替换为一个单独的关系变量 POI，带有属性 PNO、ON_SALE 以及 IN_STOCK（其中属性 ON_SALE 和 IN_STOCK 的类型是 BOOLEAN 并且拥有明显的解释说明，而{PNO}则是单键）。那么这个关系变量的一个可能的值如图 11.2 所示（与第 9 章的图 9.3，以及第 10 章中的图 10.4 都是一致的）。

POI

| PNO | ON_SALE | IN_STOCK |
|---|---|---|
| P1 | TRUE | FALSE |
| P2 | TRUE | TRUE |
| P3 | TRUE | FALSE |
| P4 | FALSE | FALSE |
| P5 | FALSE | TRUE |
| P6 | TRUE | FALSE |

图 11.2　关系变量 POI 的样本值

[4] 你可能还记得在第 9 章和第 10 章中，在这里会有一个相关的问题出现：根据前两章的更新规则定义，向交集视图插入一个数组然后马上删除，或者从合并视图中删除一个数组然后马上插入回来，都可能无法保证状态不变（这里的“状态不变”指的是定义视图的关系变量的状态，而对于视图本身则状态不变可以一直保持）。这里我要再给大家留个练习，去验证对于差异视图是否有类似的问题存在。

现在我们来定义限制视图 PL’和 PK’以及差异视图 DLK’，详情如下。

```
CONSTRAINT ... PL′  = POI WHERE ON_SALE ;
CONSTRAINT ... PK′  = POI WHERE IN_STOCK ;
CONSTRAINT ... DLK′  = PL′  MINUS PK′  ;
```

图 11.3 所示的是在图 11.2 情况下这些关系变量的相应样本值。

PL′

| PNO | ON_SALE | IN_STOCK |
|---|---|---|
| P1 | TRUE | FALSE |
| P2 | TRUE | TRUE |
| P3 | TRUE | FALSE |
| P6 | TRUE | FALSE |

DLK′

| PNO | ON_SALE | IN_STOCK |
|---|---|---|
| P1 | TRUE | FALSE |
| P3 | TRUE | FALSE |
| P6 | TRUE | FALSE |

PK′

| PNO | ON_SALE | IN_STOCK |
|---|---|---|
| P2 | TRUE | TRUE |
| P5 | FALSE | TRUE |

图 11.3　关系变量 PL’、PK’以及 DLK’的样本值

这 3 个关系变量都以{PNO}作为单键，而{PNO}同时也是DLK’的一个外键，引用PL’。而下面所列的这些约束也同时起效。[5]

```
CONSTRAINT ... IS_EMPTY ( PL′  WHERE NOT ( ON_SALE ) ) ;
CONSTRAINT ... IS_EMPTY ( PK′  WHERE NOT ( IN_STOCK ) ) ;
CONSTRAINT ... IS_EMPTY ( DLK′  WHERE NOT ( ON_SALE ) OR IN_STOCK ) ) ;
```

在第 9 章中（而不是第 10 章），我们还需要一个额外的约束来实现这个效果，那就是如果某个零部件同时存在于 PL’和 PK’中，那么代表这个零部件的两个数组实际上应该是同一个。

```
CONSTRAINT ... UNION { PL′ , PK′ } KEY { PNO } ;
```

当然，如果关系变量 PL’和 PK’确实如我们所定义的是 POI 的视图的话，那么这个约束就将自动强制执行。

下面是对关系变量 POI 以及关系变量 PL’和 PK’的更新规则，与第 9 章和第 10 章所列的一样。

[5] 在第 9 章和第 10 章中，这里还有一个约束可以实现这个效果，那就是限制 PL’ WHERE IN_STOCK 和 PK’ WHERE ON_SALE 必须相等，作为关系变量 PL’和 PK’违反“正交设计原则”的后果。

```
ON INSERT i INTO POI :
   INSERT ( i WHERE ON_SALE ) INTO PL′ ,
   INSERT ( i WHERE IN_STOCK ) INTO PK′ ;

ON DELETE d FROM POI :
   DELETE ( d WHERE ON SALE ) FROM PL′ ,
   DELETE ( d WHERE IN_STOCK ) FROM PK′ ;

ON INSERT i INTO PL′ : INSERT i INTO POI ;

ON INSERT i INTO PK′ : INSERT i INTO POI ;

ON DELETE d FROM PL′ : DELETE d FROM POI ;

ON DELETE d FROM PK′ : DELETE d FROM POI ;
```

而下面这些则是关系变量 PL’和 PK’以及关系变量 DLK’的更新规则。

```
ON INSERT i INTO PL′ : INSERT ( i WHERE NOT ( IN_STOCK ) ) INTO DLK′ ;
ON INSERT i INTO PK′ : DELETE i FROM DLK′ ;

ON DELETE d FROM PL′ : DELETE d FROM DLK′ ;
ON DELETE d FROM PK′ : INSERT ( d WHERE ON_SALE ) INTO DLK′ ;

ON INSERT i INTO DLK′ : INSERT i INTO PL′ ;

ON DELETE d FROM DLK′ : DELETE d FROM PL′ ;
```

我相信大家也都认可，上面这些规则都是毫无争议的（尽管在第 9 章和第 10 章中，我认为把它们与以前的版本的例子做比较是很有意义的）。这里引用第 9 章和第 10 章的话：实际上，我所做的其实是重新设计了数据库，把原来只能通过 PL 和 PK 隐式表达的信息变为现在可以由属性 ON_SALE 和 IN_STOCK 的值显式表达。而这种重新设计的效果就是将上一个版本的例子中挣扎于那些差强人意的更新行为的情况，转变为一个更新行为更可接受的版本。

我将把当用户只能看到关系变量 DLK’时对上述所有内容的验证留给大家作为练习。另外，我要再给大家留个练习， DLK’可以被直接定义为 POI 的限制条件——只要我们想，那么请大家验证一下在这种情况下的关于更新的限制条件是否与上面讲的差异视图的版本在本质上是一致的。最后，我还要给大家留一个练习，假设系统是可以支持相应的更新操作的，那么请大家证明这个例子的“双引号”版本是完全不可行的，这个版本类似于第 9 章示例 2 的那个“双引号”版本。

### 11.1.2 关于“包含差异”

下面讨论一个略有差异的例子。假设我们给定两个关系变量，PL（和上面一样给

出的是正在销售的零部件号码）以及 PD（给出在国内生产的零部件号码），并且实际上这里有一条商业规则，规定只有在国内生产的零部件才能够销售。由于这条规则，在任何时间，由当前 PL 值确定的关系都被包含于由 PD 当前值确定的关系中。当然，对于差异 DDL= PD MINUS PL 给出的是在国内生产，但却没有在销售的零部件号码。（请注意，这个 DDL 是一个“包含差异”的例子，因为第 2 个运算数关系变量 PL 被包含在第一个当中。更具体来说就是在这个命题的情况下，PL 的关系体在任意给定时间都是 PD 的关系体的子集。）各个关系变量的谓词如下所示。

PL：零部件 PNO 正在销售。

PD：零部件 PNO 是在国内生产的。

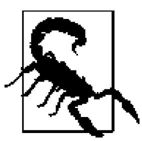
DDL：零部件 PNO 是在国内生产并没有在销售的。

下面我们来看看，对比本节之前我们所讨论的涉及到关系变量 PL 和 PK 的例子来说，这个例子最重要的一点，那就是下面这个约束显然起效。

```
CONSTRAINT ... PL ⊆ PD ;
```

而这个约束所带来的结果就是对于在本节第一部分所给定的那些补偿性操作来说，对 DDL 进行的删除操作总会因为违反黄金法则而失败。

## 11.2 示例 2：显式重叠



我将本节包含进来是为了文章的完整性。但其实里面并没有加入太多内容，你大可以直接跳过本节阅读。

下面，让我们来看另一个例子，它基于大家现在都已经很熟悉的关系变量 NLS（“non London suppliers”）和 NPS（“non Paris suppliers”）。

```
NLS { SNO , SNAME , STATUS , CITY } KEY { SNO }
NPS { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

下面再为大家列出它们的谓词。

NLS：供应商 SNO 已经签约，名字是 SNAME，拥有状态 STATUS，并且位于城市 CITY（CITY 不是 London）。

NPS：供应商 SNO 已经签约，名字是 SNAME，拥有状态 STATUS，并且位于城市 CITY（并且 CITY 不是 Paris）。

接下来让我们给出差异视图 DLP 的定义 NLS MINUS NPS。

```
DLP { SNO , SNAME , STATUS , CITY } KEY { SNO }
```

该关系变量的谓词是：

DLP：供应商SNO已经签约，名字是SNAME，拥有状态STATUS，并且位于城市CITY（CITY是Paris）。[6]

样本值如图 11.4 所示。

NLS

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S5 | Adams | 30 | Athens |

DLP

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |

NPS

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

图 11.4　关系变量 NLS、NPS 以及 DLP 的样本值

下列约束显然起效。

```
CONSTRAINT ... DLP = NLS MINUS NPS ;
CONSTRAINT ... IS_EMPTY ( NLS WHERE CITY = 'London' ) ;
CONSTRAINT ... IS_EMPTY ( NPS WHERE CITY = 'Paris' ) ;
CONSTRAINT ... IS_EMPTY ( DLP WHERE CITY ≠ 'Paris' ) ;
CONSTRAINT ... ( NLS WHERE CITY ≠ 'Paris' ) =
                         ( NPS WHERE CITY ≠ 'London' ) ;
```

同时，如前所述，这 3 个关系变量各自都以{SNO}为单键，而在 DLP 中{SNO}是个外键，引用 NLS。注意：与之前一样，我们还需要一个额外的约束来使得{SNO}是 NLS 与 NPS 合并的键，从而确保如果某个供应商同时在两个关系变量中，那么代表那个供应商的两个数组其实就是同一个。

```
CONSTRAINT ... UNION { NLS , NPS } KEY { SNO } ;
```

[6] 严格地讲，在括号中的这部分谓词应当读作“并且 CITY 不是 London，而且不是（不是 Paris）。”不过很显然后来这个描述简化成了“并且 CITY 是 Paris”。由此，DLP 和 NPS 相互之间的更新行为应当与 LS 和 NLS 相互之间的更新行为（参见第 4 章）一样，当然其中涉及到 London 的地方必须都换成 Paris。注意：相对而言，NPS MINUS NLS 的谓词是“供应商 SNO 已经签约，名字是 SNAME，拥有状态 STATUS，并且位于城市 CITY（CITY 是 London）。”

当然，如果关系变量 NLS 和 NPS 确实是供应商关系变量 S 的视图，那么这条约束就会自动强制执行。

下面要说的是补偿性操作。首先，是常见的 NLS 和 NPS 相互之间的级联更新规则。

```
ON INSERT i INTO NLS : INSERT ( i WHERE CITY ≠ 'Paris' ) INTO NPS ;
ON INSERT i INTO NPS : INSERT ( i WHERE CITY ≠ 'London' ) INTO NLS ;

ON DELETE d FROM NLS : DELETE ( d WHERE CITY ≠ 'Paris' ) FROM NPS ;
ON DELETE d FROM NPS : DELETE ( d WHERE CITY ≠ 'London' ) FROM NLS ;
```

接下来是从 NLS 到 DLP 的级联插入规则。

```
ON INSERT i INTO NLS : INSERT ( i WHERE CITY = 'Paris' ) INTO DLP ;
```

以及从 NLS 到 DLP 的级联删除规则。

```
ON DELETE d FROM NLS : DELETE ( d WHERE CITY = 'Paris' ) FROM DLP ;
```

（在这里我们可以无损地去掉限制条件。）

至于对 NPS 的插入规则，在 11.1 节的分析应该会告诉你我们需要形如下面的规则。

```
ON INSERT i INTO NPS : DELETE i FROM DLP ;
```

但是如果向 NPS 中插入数组 t，那么数组 t 的 CITY 值必须是 Paris 以外的地方，而 DLP 中所有数组的 CITY 值全部都是 Paris，因此这里的补偿性操作就变为了“空操作”。

那么最后，对 NPS 的删除规则又是怎样的呢？同样，在 11.1 节的分析应该会告诉你我们需要形如下面的规则。

```
ON DELETE d FROM NPS : INSERT ( d INTERSECT NLS ) INTO DLP ;
```

但是NPS中所有数组的CITY值都不是Paris，因此所有d的数组也是如此，而所有d INTERSECT NLS的数组则更是如此。[7]既然DLP中数组的CITY值都是Paris，那么补偿性操作在这里就再次成为了“空操作”。注意：实际上，不严谨的讲，将这两个“空操作”放在一起，我们能够很直观地发现在NPS（=non Paris suppliers）上进行插入和删除操作其实对DLP（=Paris suppliers）不会有任何影响。

至于对差异关系变量本身的更新，通过与上面很类似的分析我们很容易看出，规则简化到下面这种形式。

```
ON INSERT i INTO DLP : INSERT i INTO NLS ;
ON DELETE d FROM DLP : DELETE d FROM NLS ;
```

[7] 请大家再次回想一下第 4 章中所讲的，在 d 中的每一个数组也都会出现在 NPS 中。

这两个规则将我们带到了示例 2 的末尾。不过可能现在大家可以看出为什么我要先和大家讨论示例 1“零部件正在销售”对“零部件有库存”这个例子了，因为示例 2，NLS 对 NPS 这个例子，虽然有比示例 1 更好的表现，但并不是一个能够对底层概念进行很好诠释的好范例，这正是因为一些特定约束的存在导致更新规则变成了特别简化的形式（让人有点一叶障目的感觉）。

## 11.3 小结

让我们从 11.2 节的示例讨论中抽离出来，并总结一下遇到差异变量时的更新规则吧。定义 V 为 A MINUS B。然后，为了简单易懂，我们假设所有要被插入到 A 和 B 中的数组都必须分别满足布尔表达式 ax 和 bx，这里 ax 和 bx 代表限制条件（简单默认为 TURE）。于是我们有：

```
ON INSERT i INTO A :
   INSERT ( i WHERE NOT ( bx ) ) INTO V ;
ON INSERT i INTO B :
   DELETE i FROM V ;
ON DELETE d FROM A :
   DELETE d FROM V ;
ON DELETE d FROM B :
   INSERT ( d WHERE ax ) INTO V ;
ON INSERT i INTO V :
   INSERT i INTO A , DELETE i FROM B ;
ON DELETE d FROM V :
   DELETE d FROM A , INSERT d INTO B ;
```

我将再次把以下验证留给大家作为练习：本章前面例子的更新规则都是上述这些通用规则的特殊情况，有些可能是稍微“改进”的情况。

现在，我们看到，对于示例 1（隐式重叠的例子），通过一个差异视图进行删除操作的规则在实际操作中有时候会导致结果不可接受，或者至少不很理想。让我来详细说明一下这一点。令A和B的谓词分别为PA和PB，那么对于V= A MINUS B的谓词就应当是PA AND NOT (PB)。于是，如果我们从V中删除t，那么就意味着命题PA(t) AND NOT (PB(t))为假，这样就逻辑上等价于说NOT(PA(t)) OR PB(t)为真。因此如果从V中删除t导致t被从A中删除，而被插入B中，那么我们等于再次将逻辑OR与逻辑AND等同对待。[8]

我们同样可以看到，上述的结论适用于“零部件正在销售”对“零部件有库存”的

[8] 我要再次提醒大家，David McGoveran 有一个办法来解决歧义问题，我将在第 15 章中进行讨论。

第 1 个版本的例子。然而，真正的问题在于给定一个特定的数组并将其从差异 DLK 中删除，数据库管理系统无法判断该数组逻辑上应当属于关系变量 PK 或不属于。实际上，在该例子中，对于关系变量 PK 相应的限制条件就是简单的 TURE。而我对于这个问题的解决方案实际上就是改变数据库设计，从而让数据库管理系统最终可以判断给定的数组到底应该属于哪个（些）关系变量。我推荐的这个方案在实际操作中将会很有用，并且除了视图更新本身以外，在其他领域也能够提供很多便利。

# 第 12 章

# 组合视图和去组合视图

更新组，不需要循环

——Anon：《Where Bugs Go》

在这一章中我将与大家讨论通过关系组合和去组合运算符（在 Tutorial D 中是 GROUP 和 UNGROUP）来进行更新的问题。鉴于大家可能对这两个操作符不是很熟悉，我将先给大家做一个简要介绍。

## 12.1　组合（GROUP）和去组合（UNGROUP）运算符

让我们来看看图 12.1 中所示的关系，我们可以分别找到两个关系变量 SP 和 SPQ 的当前值。当然，关系变量 SP 就是我们常用的供应商-零部件数据库中的那个设备供应关系变量。

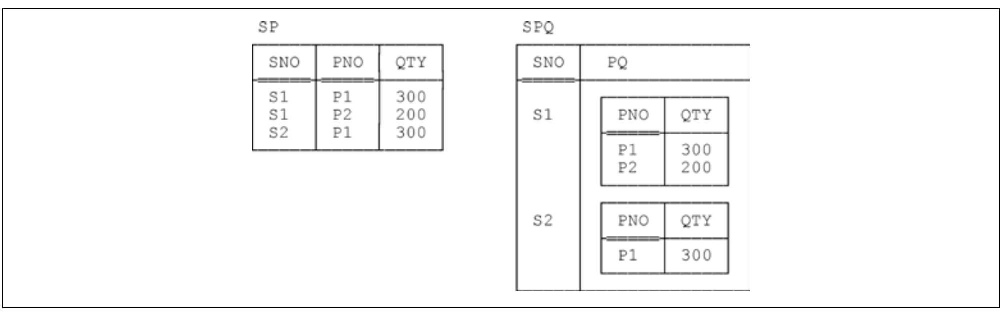



图 12.1　关系变量 SP 以及 SPQ 的样本值

那么，至少直观上看起来我们应当能感觉到这里再次出现了信息等价[1]（因为图 12.1 中所示的各关系表达了完全相同的信息）。那么关于这两个关系，图中左边的这一个就是通常我们看到的关系变量SP样本值的缩减版，而右边这一个则是我们将SP样本值带入下面的表达式从中得出的。

```
SP GROUP ( { PNO , QTY } AS PQ )
```

作者注：实际上这里的GROUP调用逻辑上等价于一个特定的EXTEND调用。具体来说就是这个调用EXTEND SP{SNO}:{PQ := !!SP},where !!SP表示一个特定的“映像关系”[2]，而我们有理由选择这个EXTEND公式，而不是与它相同的GROUP。不过关于这一点的细节讨论已经超出了本书的范围。如果大家还想了解更多关于这个问题的解释，请参看《SQL and Relational Theory》。

现在，图 12.1 中右边的关系应当就是关系变量 SPQ 的当前值，如果我们将其代入下面所列的表达式求值。

```
SPQ UNGROUP ( PQ )
```

那么就会得到图中左边的关系。关于术语的解释：图表中右侧的关系是左侧关系的一个“组合后”的版本（具体来说是将PNO和QTY进行组合），而左侧的关系则是右侧关系的一个“去组合后”的版本（具体来说就是将PQ去组合）。属性PQ是一个关系值属性（relation valued attribute，RVA）[3]。

让我们回过头来再看一下关系变量 SPQ。下面是该关系变量在 Tutorial D 下的定义。

```
VAR SPQ BASE RELATION
  { SNO CHAR , PQ RELATION { PNO CHAR , QTY INTEGER } }
    KEY { SNO } ;
```

请大家特别注意观察属性 PQ 拥有特定的关系类型，正如我说过的该属性是具有关系值的。注意：和往常一样，该关系变量显然受制于各种约束，在这里我要特别指出一种，这个约束就是{SNO}是该关系变量的单键，但现在我并不想针对它做过多讨论，以免跑题。

下面，让我来正式地给 GROUP 和 UNGROUP 下个定义。不过为了能够更好地对它们进行定义，我要首先向大家介绍另外两个附加的运算符，分别是 WRAP 和 UNWRAP。总地来说，WRAP 将给定关系中具体的一些属性包裹成一个独立的属

[1] 不过到本节结尾时，请注意参看注脚。
[2] 我们在第 8 章中曾经提到过映像关系。
[3] 我们在第 2 章中曾经提到过 RVA。

性具有数组值，而 UNWRAP 则进行逆向操作。例如，在图 12.1 中左边的关系是关系变量 SP 的当前值，那么下列表达式：

```
SP WRAP ( { PNO , QTY } AS PQ )
```

会得出下列关系：

<table>
<tr><th>SNO</th><th>PQ</th></tr>
<tr><td>S1</td><td><table><tr><th>PNO</th><th>QTY</th></tr><tr><td>P1</td><td>300</td></tr></table></td></tr>
<tr><td>S1</td><td><table><tr><th>PNO</th><th>QTY</th></tr><tr><td>P2</td><td>200</td></tr></table></td></tr>
<tr><td>S2</td><td><table><tr><th>PNO</th><th>QTY</th></tr><tr><td>P1</td><td>300</td></tr></table></td></tr>
</table>

注意这里 PQ 的值在这个关系中都是数组，而不是关系。也就是说，PQ 在这里是具有数组值的属性，而不是具有关系值的属性。另外还请注意以下几点：（a）属性 PQ 具有双下画线（如果这个关系被理解为某个关系变量的当前值，那么该关系变量的单键就是整个关系头）；（b）属性 PNO 在 3 个 PQ 值中并“没有”双下画线，这正是因为这些 PQ 值都是数组，而不是关系，而键只适用于关系，而非数组。

至于 UNWRAP 我们这样来看，让我们把上述 WRAP 操作的结果称作 spw，那么下列表达式：

```
spw UNWRAP ( PQ )
```

返回的结果将是最开始的设备供应关系（就是图 12.1 中左侧含有 3 个数组的关系）。

下面，我将正式给出 GROUP 和 UNGROUP 的定义。注意：这里我给出这些定义的主要目的是参考及引用。像很多正式的定义一样，在第一次阅读的时候它们可能有点难懂。所以我认为跟大家提前说明这一点很重要，那就是在这个阶段我们不需要百分之百理解这些定义。

- **定义：**令关系 r 的关系头被分解为子集 X = {X1, X2, ..., Xm}以及 Y = {Y1, Y2, ..., Yn}，同时，令 YR 作为一个不出现在 X 中的属性的名字。那么表达式 r GROUP

({Y1, Y2, ..., Yn} AS YR)会返回一个关系 s。s 的关系头为{X1, X2, ..., Xm, YR}，这里 YR 的类型是 RELATION {Y1, Y2, ..., Yn}。S 的关系体定义如下。令 r WRAP ({Y1, Y2, ..., Yn} AS YT)的结果为 z。对于 z 中每一个不同的 X 的值 x，令 yr 成为这样一个关系——其数组全部都是且仅包含 z 中数组的 YT 值，z 中的 X 值为 x，令 t 为类型为 TUPLE {X, YR}的数组，其中含有 X 的值 x 以及 YR 的值 yr，那么只有在这种情况下，t 是 s 的一个数组。

- **定义：**令关系 s 拥有一个类型为 RELATION {Y1, Y2, ..., Yn}的属性 YR，并令 X= {X1, X2, ..., Xm}为除 YR 之外所有的属性，同时，令所有 Xi 都不与任何 Yj 重名(0≤i≤m, 0≤j≤n)。那么表达式 s UNGROUP (YR)将返回一个关系 r。r 的关系头为{X1, X2, ..., Xm, Y1, Y2, ..., Yn}。S 的关系体定义如下。令关系 z 的关系头为{X1, X2, ..., Xm, YT}，这里 YT 的类型为 TUPLE {Y1, Y2, ..., Yn}，于是关系体有如下定义：对于 s 的每一个数组，z 都包含一组类型为{X, YT}的数组，我们设其中 t 对应在 YR 值下每一个 s 的数组，每一个这种数组 t 都包含一个 X 值等于相应 s 数组中的 X 值，以及一个 YT 值等于某些相应 s 数组中的 YR 值。令 z 不含有任何其他数组。那么 r 为 z UNWRAP (YT)的结果。

对于给定的关系r和一些r的组合，这里总会有一个逆向的去组合再次生成r，然而反过来却并不一定是这样。接下来的例子说明了这一点。我们来看一个关系，姑且叫它spq，如图 12.2 所示，请特别注意供应商S2 的PQ值在spq中是个空关系。表达式spq UNGROUP (PQ)会产生一个关系，姑且叫它sp，如图中间部分所示。而表达式sp GROUP ({PNO,QTY} AS PQ)则产生如图右侧所示的关系，该关系显然与原来的关系spq不相等。[4]

| SNO | PQ |
|---|---|
| S1 | PNO / QTY: P1 300; P2 200 |
| S2 | PNO / QTY: (empty) |

| SNO | PNO | QTY |
|---|---|---|
| S1 | P1 | 300 |
| S1 | P2 | 200 |

| SNO | PQ |
|---|---|
| S1 | PNO / QTY: P1 300; P2 200 |

图 12.2　UNGROUP 与 GROUP 不是一定互为逆运算

[4] 不过你是否认为它们信息等价呢？（在大家读过第 14 章之后可能会想要重新审视这个问题。）

## 12.2 一个组合/去组合示例

为了更好地研究对于组合和去组合关系变量的更新，让我们首先从设备供应关系变量 SP 开始，并且让我们将该关系变量针对 PNO 和 QTY 进行组合操作来定义另一个关系变量 SPQ，我们将该变量中组合后的属性称作 PQ（与 12.1 节中的基本相同）。同时，为了意义明确，我们假设在进一步说明之前，关系变量 SP 和 SPQ 都是基础关系变量，暂时没有视图。那么谓词如下所示。

SP：供应商 SNO 供应零部件 PNO，数量为 QTY。

SPQ：供应商SNO当且仅当对集合PQ包含一个数对（PNO，QTY），其中PNO等于p，而QYT等于q时，供应零部件p，数量为q。[5]

那么关于约束，{SNO,PNO}为 SP 的单键，而{SNO}则为 SPQ 的单键。同时，下列约束也显然起效。

```
CONSTRAINT ... SPQ = SP GROUP ( { PNO , QTY } AS PQ ) ;
CONSTRAINT ... SP = SPQ UNGROUP ( PQ ) ;
```

在这里我们的组合操作是具有可逆性的，因为 SPQ 是由 SP 而不是其他方式定义的，也就是说在 SPQ 中的所有数组与其 PQ 值一样都不可能为空关系。

同样地，在 SPQ 中，两个含有不同 PQ 值的数组不可能拥有一个相同的 PNO 值。实际上，这个约束是下面两个条件放在一起的逻辑结果：{SNO,PNO}是 SP 的键，同时对属性 PNO 的组合已经完成了（一部分）。不过我们依然可以对这条约束做显式声明。

```
CONSTRAINT ... ( SPQ UNGROUP ( PQ ) ) KEY { SNO , PNO } ;
```

最后，我们显然还有下面这条约束。

```
CONSTRAINT ... SPQ { SNO } = SP { SNO } ;
```

（虽然这条约束实际上也是我们如此定义 SPQ 的逻辑结果之一）。

下面，让我们来看看对关系变量 SP 的更新以及它在关系变量 SPQ 上的补偿性操作吧。为了明确表述，我们令关系变量 SP 和 SPQ 的当前值如图 12.1 所示。那么假

[5] 如大家所见，RVA 让关系变量 SPQ 的谓词显得有点复杂。这种情况也从侧面反映了为什么 RVA 至少在基础关系变量中通常被视为禁忌，虽然不是完全没有被使用。关于这点说太多就会跑题了，不过如果你感兴趣，欢迎参阅《Database design and Realtional Theory》以获得更多信息。

设我们向 SP 插入设备供应数组(S3,P2,200)。（这第 1 个例子的重点在于关系变量 SP 当前不含有任何关于供应商 S3 的数组，因此在关系变量 SPQ 中也没有。）那么显然，我们需要做的就是向关系变量 SPQ 中插入下列数组。

<table>
<tr><th>SNO</th><th>PQ</th></tr>
<tr><td>S3</td><td><table><tr><th>PNO</th><th>QTY</th></tr><tr><td>P2</td><td>200</td></tr></table></td></tr>
</table>

那么相对的，如果假设我们向 SP 中插入设备供应数组(S1,P3,400)（这第 2 个例子的重点在于关系变量 SP 当前含有关于供应商 S1 的一些数组，因此在关系变量 SPQ 中也有，当然实际上它只含有一个这样的数组），那么显然，我们需要做的就是将关系变量 SPQ 中已经存在的关于供应商 S1 的数组替换为下列数组。

<table>
<tr><th>SNO</th><th>PQ</th></tr>
<tr><td>S1</td><td><table><tr><th>PNO</th><th>QTY</th></tr><tr><td>P1</td><td>300</td></tr><tr><td>P2</td><td>200</td></tr><tr><td>P3</td><td>400</td></tr></table></td></tr>
</table>

那么上面这些考量最终让我们得出了下列对于 SP 的插入规则。

```
ON INSERT i INTO SP :
   DELETE ( SPQ MATCHING i ) FROM SPQ ,
   INSERT ( ( SP MATCHING i { SNO } )
                       GROUP ( { PNO , QTY } AS PQ ) ) INTO SPQ ;
```

针对两个或两个以上独立赋值时会有同一个目标这种情况，多重赋值专门有相关的定义（请参看附件 A），在这里我可以指望对 SPQ 的删除操作会先于对其的插入操作被执行。不过，我们也可以通过将规则改造为下面这种方式来避免任何这种对定义的依赖。

```
ON INSERT i INTO SP :
   WITH ( t1 := SPQ MATCHING i ,
          t2 := SP MATCHING i { SNO } ,
          t3 := t2 GROUP ( { PNO , QTY } AS PQ ) ) :
   DELETE t1 FROM SPQ ,
   INSERT t3 INTO SPQ ;
```

关于这个例子有个很有趣的地方就是相应的删除规则与我们刚刚定义的插入规则非常相似。详细点说就是如果从 SP 中删除一个数组，那么 SPQ 中相应供应商的单独数组也需要被删除，然后该供应商在 SP 中遗留下来的数组（如果有）可以被用来计算出一个新的数组并插入到 SPQ 中。规则详情如下。

```
ON DELETE d FROM SP :
   WITH ( t1 := SPQ MATCHING d ,
          t2 := SP MATCHING d { SNO } ,
          t3 := t2 GROUP ( { PNO , QTY } AS PQ ) ) :
   DELETE t1 FROM SPQ ,
   INSERT t3 INTO SPQ ;
```

那么对于 SPQ 的更新又是怎样的呢？其中删除规则很简单。

```
ON DELETE d FROM SPQ :
   DELETE ( SP MATCHING d ) FROM SP ;
```

实际上我认为插入规则也很简单。举例来说，我们再次假定关系变量 SP 和 SPQ 的当前值是如图 12.1 所示的关系。那么大家考虑一下，如果我们将下面的数组插入到 SPQ 中会发生什么。

<table>
<tr><th>SNO</th><th>PQ</th></tr>
<tr><td>S4</td><td><table><tr><th>PNO</th><th>QTY</th></tr><tr><td>P2</td><td>200</td></tr><tr><td>P4</td><td>300</td></tr><tr><td>P5</td><td>400</td></tr></table></td></tr>
</table>

我想大家已经很清楚，我们需要做的就是将下列 3 个数组插入到关系变量 SP 中。

| SNO | PNO | QTY |
|---|---|---|
| S4 | P2 | 200 |

| SNO | PNO | QTY |
|---|---|---|
| S4 | P4 | 300 |

| SNO | PNO | QTY |
|---|---|---|
| S4 | P5 | 400 |

下面是插入规则。

```
ON INSERT i INTO SPQ :
   INSERT ( i UNGROUP ( PQ ) ) INTO SP ;
```

那么，请大家考虑一下，根据这个规则，如果我们尝试向 SPQ 中插入一个 PQ 值为空的空关系数组会出现怎样的情况？假设空关系数组如下所示。

<table><tr><th>SNO</th><th>PQ</th></tr><tr><td>S4</td><td><table><tr><th>PNO</th><th>QTY</th></tr><tr><td></td><td></td></tr></table></td></tr></table>

这里我们设 i 为包含上述数组的关系，于是对表达式 i UNGROUP (PQ)求值的结果将是一个空关系。因此，没有数组被插入到 SP 中。所以整个操作会因为违反黄金法则而失败，具体来说，它违反了效果为 SPQ{SNO} = SP{SNO}的约束。

现在让我们来回想一下，到目前为止，我一直假设 SP 和 SPQ 都是基础关系变量。但是我想你可以看到，如果 SPQ 实际上是 SP 的一个（组合）视图的话也并没有什么不同，所有我们前面讨论的更新规则都可以毫无变化地适用。反过来讲，如果 SP 是 SPQ 的一个（去组合）视图情况也是一样。更有甚者，即使我们不能保证在 SPQ 中的所有数组与其 PQ 值一样都不为空关系也没有问题，尽管在这种情况下信息等价是肯定丢失了的（因为约束 SPQ = SP GROUP ({ PNO,QTY} AS PQ)不再起效）。

## 12.3　一个 SUMMARIZE（概览）示例

Tutorial D 的 SUMMARIZE 运算符主要分为两种形式。对于更加通用的形式我将放到下一章（第 13 章）中讨论，不过，在这里讨论一下简单一点的形式却很方便。（为什么说方便呢，因为虽然这个简单版本其实是通用版本的特殊情况，不过我们依然可以把它看作前两节中讨论的 GROUP 运算符的变种。）下面我们来看一个例子。再次令关系变量 SP 的当前值为如图 12.1 所示的关系，那么对下列表达式求值：

```
SUMMARIZE SP BY { SNO } : { TQY := SUM ( QTY ) } ;
```

会得出下列关系：

| SNO | TQY |
|---|---|
| S1 | 500 |
| S2 | 300 |

这个结果是通过如下步骤得来的（我们在后面会给出正式的定义）。首先，对表达式 SP GROUP ({PNO,QTY} AS PQ)求值，可以得到一个中间结果，我们设它为 spq；接下来，对于 spq 中的每一个数组，都将其 PQ 值（当然这个值是个关系）替换为 TQY 值，而这个 TQY 值来自于对 PQ 关系的“求和”计算 SUM(QTY)。

作者注：实际上，前面我们讲的 SUMMARIZE 调用逻辑上等价于一个特定的

EXTEND 调用，具体来讲就是在 EXTEND SP{SNO}:{TQY :=SUM( !!SP,QTY)}这个调用中，!!SP 体现了一个特定的映像关系，而在这里我们有理由从二者之中选择 EXTEND 公式，而不是 SUMMARIZE。不过，关于这个问题的细节讨论已经超出了本书的范围。如果大家有兴趣请参看《SQL and Relational Theory》来获得更多说明。

接下来我们还有另一个例子（“对于每个供应商，计算每种不同的‘设备供应’数量之和”）。

```
SUMMARIZE SP { SNO , QTY } BY { SNO } : { TQY := SUM ( QTY ) }
```

这个表达式所表达的是对于SP上SNO和QTY的一个特定投影的概览，这个投影只包括对每个供应商号码相应的每种不同设备供应的数量，正如我们所要求的。[6] 注意：依照图 12.1 所给出的SP的样本值，这个表达式恰好得出了与上一个例子相同的结果，不过通常来讲不会这样。

接下来是对这个形式的SUMMARIZE的定义。[7]

- **定义：**令关系 r 的属性叫作 A1, A2, ..., An（可能还有更多），而没有属性叫作 B。那么，表达式 SUMMARIZE r BY {A1, A2, ..., An} : {B :=summary}将返回一个关系，其关系头为{A1, A2, ..., An, B}，而关系体为所有数组 t 的集合，这里的 t 等于 r 中某些数组在 A1, A2, ..., An 上的投影，并扩展出一个 B 的值 b。这个值 b 是通过对所有与 t 具备相同属性（包括 A1, A2, ..., An）值的 r 的数组进行“求和”得出的。

作者注：我认为就本书的目的而言，对“计算一个求和”这个概念可以被定义得更准确一些。[8] 不过我实在不想在这里尝试写出这样一个定义，因为这会产生相当多的复杂性，这些复杂的内容（很可能）对本书的主题没有任何容忍度。因此我就默认大家直观上关于这里发生了什么有了足够的了解并且不再深究。同时我默认大家对于一些经常会被用到的概览操作也都很熟悉了，比如计数、求和、求平均等。注完。

那么现在，让我们假设 SP 是基础关系变量，同时让我们用下面所列的约束定义一个视图 STQ。

---

[6] 从这个例子中已经能充分反应出，Tutorial D 完全不需要，也不支持类似 SQL 那个允许“概览”中包含关键词 DISTINCT 的特色小把戏。

[7] 注意观察这个定义使用了数组版的投影，同时也适用了数组版的关系运算符 EXTEND。请详见《SQL and Relational Theory》来获得关于这个问题的更多讨论。

[8] 在这个例子中，对求和运算求值 SUM(QTY)实际上需要对“PQ 关系”的属性 QTY 调用“求和运算符”SUM。关于这一点的更多解释，请参看《SQL and Relational Theory》。

```
CONSTRAINT ... STQ = SUMMARIZE SP BY { SNO } : { TQY := SUM ( QTY ) } ;
```

各变量谓词如下。

SP：供应商 SNO 供应零部件 PNO，数量为 QTY。

STQ：供应商 SNO 供应零部件总数为 TQY。

实际上 STQ 的谓词略有点站不住脚，如果有一点点改进就会很好了。在下一章中我会回到这个问题来讲一讲，不过你可能想自己试试看是否能够明白我在这里所看到的问题。

当然，在这个例子中信息等价是不存在的。这也是为什么将 STQ 定义为 SP 的视图是合理的，但是将 SP 定义为 STQ 的视图却不合理，实际上根本无法这样定义。

我们再来看看约束，{SNO,PNO}是 SP 的单键，而{SNO}是 STQ 的单键。同时，下列约束也会起效。

```
CONSTRAINT ... STQ { SNO } = SP { SNO } ;
```

而这个约束其实也是 STQ 的定义方式所带来的结果。

现在，让我们来看看在关系变量 SP 上的更新情况以及相应的补偿性操作。为了文章的明确性，我们令关系变量 SP 和 STQ 的当前值如图 12.3 所示。

SP

| SNO | PNO | QTY |
|---|---|---|
| S1 | P1 | 300 |
| S1 | P2 | 200 |
| S2 | P1 | 300 |

STQ

| SNO | TQY |
|---|---|
| S1 | 500 |
| S2 | 300 |

图 12.3　关系变量 SP 和 STQ 的样本值

假设我们向 SP 中插入设备供应数组（S3，P2，200），那么显然我们需要做的就是向关系变量 STQ 中插入数组（S3，200）。对比来看，假设我们向 SP 中插入设备供应数组（S1，P3，400），那么现在我们需要做的是将关系变量 STQ 中已经存在的关于供应商 S1 的数组替换为这一个：（S1，900）。因此对于 SP 的插入规则如下所示。

```
ON INSERT i INTO SP :
   WITH ( t1 := STQ MATCHING i ,
          t2 := SP MATCHING i { SNO } ,
          t3 := SUMMARIZE t2 BY { SNO } : { TQY := SUM ( QTY ) } ) :
   DELETE t1 FROM STQ ,
   INSERT t3 INTO STQ ;
```

所以如你所见，上述规则与 12.2 节中 GROUP/UNGROUP 例子相应的规则是非常相似的，并且很明显，删除规则也同样是非常相似的。详细来说就是如果从 SP 中删除一个数组，那么 STQ 中相关供应商的单数组也需要被删除，然后该供应商在 SP 中遗留的数组（如果有）会被用来重新计算出一个新的数组并插入到 STQ 中。下面就是删除规则。

```
ON DELETE d FROM SP :
   WITH ( t1 := STQ MATCHING d ,
          t2 := SP MATCHING d { SNO } ,
          t3 := SUMMARIZE t2 BY { SNO } : { TQY := SUM ( QTY ) } ) :
   DELETE t1 FROM STQ ,
   INSERT t3 INTO STQ ;
```

那么对于 STQ 的更新又是怎样的呢？其中删除规则还是很简单。

```
ON DELETE d FROM STQ :
   DELETE ( SP MATCHING d ) FROM SP ;
```

但是，插入操作则看起来不合理，例如，向STQ插入数组（S5，600）是什么意思呢？因此在这里对于STQ并没有插入规则。[9] 当然，没有规则并不意味着我们不能向其进行插入操作（因为SP的更新规则很明显就要求向STQ插入数组），只是说这类操作基本上会因为违反黄金法则而失败，除非这些操作是某个多重赋值的一部分（在对SP的更新规则中它们就是如此）。

[9] 实际上有时候可能我们还是可以对如 STQ 这样的视图执行插入操作。例如，这里可能只有一个被加数，或者，有些起效的约束可能会有让视图成为“可被插入”的。例如，可能某些约束会让所有被加数都相等。（这里要感谢 David McGoveran 的贡献。）

第 13 章

# 展开视图与概览视图

东拉西扯，展开和概览，全都守规矩，好学校都这么教。

——Anon：《Where Bugs Go》

关系型展开（EXTEND）和概览（SUMMARIZE）操作彼此很相似，因为它们都会生成包含一个新的“计算”属性[1]的结果。简单说来，我们可以认为EXTEND能够实现“跨数组”运算，而SUMMARIZE能够进行“全属性”运算。但是我不得不说，这个特性描述及其不准确，因为实际上SUMMARIZE可以由EXTEND来进行定义，这个问题在《SQL and Relational Theory》一书中做过解释（而且在前面的一些章节中也顺便提过）。

## 13.1 一个展开示例

我将从第 2 章例子的一个简化版开始说起。首先，为了简化示例，让我们将关系变量 P 中除了 PNO 和 WEIGHT 以外的所有属性全都忽略掉。其次，假设 WEIGHT 值在该关系中的单位是磅。再次，假设我们希望将这些重量用公克显示出来。而一磅等于 454 克，因此我们可以这样写。

```
EXTEND P : { GMWT := WEIGHT * 454 }
```

依据我们常用的样本值，结果如下图所示。

[1] 在这里我忽略掉了第 2 章中 EXTEND 的形式，我姑且称之为 EXTEND 的第二形式，在该形式中被“计算”的属性并不是一个“新的”属性。同样，我会在本章的整个章节中都忽略这种形式的 EXTEND，因为它可以直接由第一形式来定义。

| PNO | WEIGHT | GMWT |
|---|---|---|
| P1 | 12.0 | 5448.0 |
| P2 | 17.0 | 7718.0 |
| P3 | 17.0 | 7718.0 |
| P4 | 14.0 | 6356.0 |
| P5 | 12.0 | 5448.0 |
| P6 | 19.0 | 8626.0 |

接下来让我们同往常一样假设关系变量 P 是基础关系变量，同时让我们定义一个如上面展开版本的视图，叫作 PX。那么各变量的谓词如下所示。

P：零部件 PNO 在公司内使用，并且重量为 WEIGHT 磅。

PX：零部件 PNO 在公司内使用，并且重量为 WEIGHT 磅和 GMWT 克（并且 GMWT = WEIGHT×454）。

而约束的情况则是{PNO}分别为 P 和 PX 的单键。同时，我们显然有以下约束起效。

```
CONSTRAINT ... PX = EXTEND P : { GMWT := WEIGHT * 454 } ;
CONSTRAINT ... IS_EMPTY ( PX WHERE GMWT ≠ WEIGHT * 454 ) ;
```

那么，在关系变量 P 上的更新可以直接进行。补偿性操作如下所示。

```
ON DELETE d FROM P :
   DELETE ( PX MATCHING d ) FROM PX ;

ON INSERT i INTO P :
   INSERT ( EXTEND i : { GMWT := WEIGHT * 454 } ) INTO PX ;
```

对于 PX 的更新也可以直接进行。

```
ON DELETE d FROM PX :
   DELETE ( P MATCHING d ) FROM P ;

ON INSERT i INTO PX :
   INSERT i { PNO, WEIGHT } INTO P ;
```

注意，观察最后一条规则，所有 i 的数组必须都有 GMWT =WEIGHT×454（否则就会违反黄金法则）。换句话来说就是只由关系变量 P 组成的设计，以及只由关系变量 PX 组成的设计，这两种设计是信息等价的。但是它们能够如此，都是因为下面这一条约束起效：PX 中的所有数组都遵守 GMWT = WEIGHT×454。如果没有这条约束，我们就可以向 PX 插入数组（例如）WEIGHT = 12.0 并且 GMWT = 0.0，

而这将是一条在 P 中找不到对应项的对 PX 的更新。

现在我们来考虑一个用户只能看到关系变量 PX 时的情况，该用户具有以下特点。

1．知道下列谓词。

PX：零部件 PNO 在公司内使用，并且重量为 WEIGHT 磅和 GMWT 克（并且 GMWT = WEIGHT×454）。

2．了解{PNO}是该关系变量的键，同时也了解下列约束。

```
CONSTRAINT ... IS_EMPTY ( PX WHERE GMWT ≠ WEIGHT * 454 ) ;
```

3．并不知道任何补偿性操作（因为该用户并不知道关系变量 P 的存在）。

很明显，从这位用户的角度来看，对于 PX 的所有更新都可以如预期的一样进行。也许，它还不是“完全的”那么明显，也许我还需要对显式 UPDATE 操作做一些说明。重点是这一条：这类操作当然都必须符合约束 GMWT = WEIGHT×454。由此导致的结果是 WEIGHT 和 GMWT 必须总是保证同步更新，下面是个例子。

```
UPDATE PX WHERE PNO = ‘P1’ : { WEIGHT := 17.0 , GMWT := 7718.0 } ;
```

作者注：在这里我们可能会陷入一个对“虚拟”或“计算”属性的讨论。具体来讲，我们可以说在关系变量PX中的属性GMWT就是这样一种属性，那么进而我们可以说这类属性不能被直接更新，也就是说维护它们是数据库管理系统DBMS的事情。然而，请注意下面两点：（a）这类属性在基础关系变量中与在视图中的意义是相同的 （b）计算属性这个概念本身跟我们所要讲的视图更新毫无关系。（不过话说回来，它确实与可控的冗余、扩展更新、多变量约束[2]，以及补偿性操作等概念有着千丝万缕的联系。尽管如此，现在我还是决定应用“谨慎设计原则”并单纯地忽略计算属性出现的可能性[3]。）

## 关系常数

上面就是关于EXTEND本身我想说的所有内容了，不过我还想用上面EXTEND的例子来阐明另外一个比较常见的重点内容。假设我们定义一个“有名称的关系常数”，

---

[2] 实际上在本例中并没有出现多变量约束的情况。

[3] 谨慎设计原则规定：给定两个设计选项 A 和 B，只能二选一，其中 A 向上兼容 B，而 B 的含义尚未完全清楚，那么谨慎地决定为选择 A。下面我们举个 SQL 的例子。SQL 的最初设计者曾面临这样一个选择，选项 A 是禁止重复行，而选项 B 为允许重复行。而对这个问题谨慎的决定应当是选择 A——禁止重复行，这样当未来出现清晰需求要求支持重复行时，它们也可以被支持。然而很不幸，设计者们选择了 B——允许重复行。当然，这个决定后来看是非常错误的，但现在已经没有一个兼容的办法能够让 SQL 回到 A 选项的情况了。

如下所示[4]。

```
CONST WG INIT ( RELATION { TUPLE { WEIGHT 11.0 , GMWT 4994.0 } ,
                           TUPLE { WEIGHT 12.0 , GMWT 5448.0 } ,
                           TUPLE { WEIGHT 13.0 , GMWT 5902.0 } ,
                           TUPLE { WEIGHT 14.0 , GMWT 6356.0 } ,
                           TUPLE { WEIGHT 15.0 , GMWT 6810.0 } ,
                           TUPLE { WEIGHT 16.0 , GMWT 7264.0 } ,
                           TUPLE { WEIGHT 17.0 , GMWT 7718.0 } ,
                           TUPLE { WEIGHT 18.0 , GMWT 8172.0 } ,
                           TUPLE { WEIGHT 19.0 , GMWT 8626.0 } ,
                           TUPLE { WEIGHT 20.0 , GMWT 9080.0 } ;
```

由这个具体的关系常数值所构成的关系实际上相当于一个以磅计的重量与以克计的重量之间的相互转换表，其范围是 11~20 磅之间的所有整数重量。现在假设我们为我们常用的零部件关系和这个关系计算一个连接 JOIN {P,WG}。结果将会与我在原来的 EXTEND 例子中所展示的结果完全一样。

| PNO | WEIGHT | GMWT |
|---|---|---|
| P1 | 12.0 | 5448.0 |
| P2 | 17.0 | 7718.0 |
| P3 | 17.0 | 7718.0 |
| P4 | 14.0 | 6356.0 |
| P5 | 12.0 | 5448.0 |
| P6 | 19.0 | 8626.0 |

这个例子实际上向我们展示了这样一个情况，那就是在某种特定情况下，JOIN和EXTEND实际上是相同的操作。不过这并不是我想在这里强调的。我要强调的是上面这个连接实际上是个“一对多”的连接（简单来说就是WG中的一个数组与P中的许多数组连接[5]），因此我在第 8 章给大家展示的针对这类连接的更新规则就应该在这里适用。然而，在第 8 章以及在这里之前本书的所有内容中，我都默认假设视图总是由具体的关系变量定义，而非关系常数，因此每一个关系操作中的每一个关系操作数都应该是可更新的。而显然，让一个常数成为可更新的变量是很荒唐的，实际上我们可以用“矛盾”这个词。所以我之前所提出的关于各类关系操作更新的规则看来需要做一些改进或者转换，从而能够妥善解决关系常数的问题。但是，我并不想在这里岔开话题，也不想在本书中就这个问题再做更多讨论。对于现在来说，如果你应用第 8 章的更新规则来更新一个一对多的连接时忽略对“一”这一端的级联，那么你得到的本质上就是我在本节

[4] 一个有名称的关系常数就如同一个视图，只不过它的值永远不变（也因此它并不能算是一个真正的关系变量）。请大家参看《SQL and Relational Theory》以获得更多解释说明。

[5] 当然相同的，它也是一个多对一的连接，P 中的许多数组与 WG 中的一个数组连接。

前面所给出的通过EXTEND更新的规则。

## 13.2　另一个概览示例

请大家回忆一下第 12 章中出现过的一个 SUMMARIZE 表达式。

```
SUMMARIZE SP BY { SNO } : { TQY := SUM ( QTY ) } ;
```

套用我们对关系变量 SP 常用的样本值，例如图 1.1 所示的值，那么对这个表达式求值如下。

| SNO | TQY |
|---|---|
| S1 | 1300 |
| S2 | 700 |
| S3 | 200 |
| S4 | 900 |

很显然，在我们的结果中有一个问题，那就是：没有关于供应商 S5 的数组。当然，这里并没有什么神秘的地方，没有输出 S5 数组的原因是没有任何关于 S5 的数组在输入端，即关系变量 SP 的当前值。然而，如果无论怎样在结果中都有一个 S5 的输出，从感觉上来讲会更好，比如：

| SNO | TQY |
|---|---|
| S1 | 1300 |
| S2 | 700 |
| S3 | 200 |
| S4 | 900 |
| S5 | 0 |

那么这个结果就是下面的表达式所能达成的效果，这个表达式应用了我在第 12 章中提到的更通用的 SUMMARIZE 形式。

```
SUMMARIZE SP PER ( S { SNO } ) : { TQY := SUM ( QTY ) }
```

如你所见，第 1 个例子中的BY项已经被替换为一个形如PER(r)的项，在这里r的关系头是（也必须是）关系SUMMARIZE的某个子集。（在该例子中，关系SUMMARIZE是关系变量SP当前值所决定的关系。）同时关系r是该操作的驱动核心，也就是说输出的结果中一定会包含一个能够反映关系r中每个数组情况的数组。在该例子中在输出端会出现 5 个数组，因为对关系变量S在SNO上的投影（r）中有 5 个数组。特别是输出的结果中将会包含一个针对供应商S5 的数组，因为在该投影中有一个供

应商S5 的数组，不过如你所见其对应的TQY值为零[6]。练习：到底为什么对应供应商S5 的TQY值会为零？[7]

于是我们有一个对于这种形式的 SUMMARIZE 的定义（我基于文章目的进行了轻微的简化）。

- **定义**：令关系 r1 和 r2 有如下情况，r2 的关系头是 r1 关系头的某个子集。令 r2 有属性 A1，A2，…，An，除此之外无其他属性（尤其是没有属性叫作 B）。那么表达式 SUMMARIZE r1 PER (r2) : {B := summary}就会返回一个关系，其关系头为{A1, A2, ..., An, B}，而关系体为所有数组 t 的集合（该 t 是 r2 的数组），并且由属性 B 的值 b 扩展。值 b 是经由"summary"对所有 r1 的数组求值得来的，拥有与属性 A1，A2，…，An 相同的值，这点与 t 一样。

作者注：重复一下第 12 章的内容，下面有一个对简化形式的 SUMMARIZE 的样本调用。

```
SUMMARIZE SP BY { SNO } : { TQY := SUM ( QTY ) } ;
```

这个调用实际上被定义为与下面更通用的形式等价，并且可以被作为其简化版来看。那么更通用版形式的调用如下。

```
SUMMARIZE SP PER ( SP { SNO } ) : { TQY := SUM ( QTY ) } ;
```

换句话说，如果在定义中对应 r2 的关系实际上是对应 r1 关系的投影，而不仅仅是某个关系头与其某些投影的关系头相等（这也是定义的要求）的话，那么我们就可以将 SUMMARIZE 中的具体的项 PER (*r2*)由项 BY {A1, A2, ..., An}来代替，其中 A1，A2，…，An 都是 r2 的属性。

接下来，让我们再一次假设 S 和 SP 都是基础关系变量，同时让我们定义一个视图 STQ，由下面的约束表示。

```
CONSTRAINT ... STQ = SUMMARIZE SP PER ( S { SNO } ) :
                                      { TQY := SUM ( QTY ) } ;
```

[6] 如果套用我们在第 12 章中讨论过的简化形式的 SUMMARIZE，那么这个 SUMMARIZE 调用实际上就逻辑等价于一个特定的 EXTEND 调用，具体来说就是调用 EXTEND S{SNO}:{TQY := SUM(!!SP,QTY)}，其中!!SP 表示一个特定的映像关系，并且我们有理由认为这个 EXTEND 表达式优于与之等价的 SUMMARIZE 表达式。不过，我还是要再次说明，关于这个问题的细节讨论已经超出了本书的主旨，因此如果大家对这个问题感兴趣，请参阅《SQL and Relational Theory》以获取更多的解释说明。

[7] 答案如下：因为，如《SQL and Relational Theory》一书中所解释的，一个求和运算中如果没有被加数，那么结果就是零（如果是 SQL 的话，那么结果应该是"null"）。

STQ的谓词如下所示[8]。

STQ：供应商 SNO 供应的零部件总数量为 TQY。

由于套用了第 13 章的 SUMMARIZE 例子，这里并不存在信息等价。这也是为什么将 STQ 定义为 S 和 SP 的视图是合理的，但是如果反过来将 S 和 SP 定义为 STQ 的视图则显然不合理（实际上，这个定义至少在通常情况下也无法实现）的原因。

那么关于约束，{SNO,PNO}是 SP 的单键，而{SNO}是 S 和 STQ 的单键。同时，下列约束也显然起效。

```
CONSTRAINT ... STQ { SNO } = S { SNO } ;
```

当然，这个约束是 STQ 如此定义的逻辑结果。更重要的是请注意它与第 12 章中 SUMMARIZE 例子的不同点：在那个例子中，STQ 中每一个的供应商号码同时也会出现在设备供应关系变量 SP 中，而在这个例子中则是 STQ 中的每一个供应商号码同时也会出现在供应商关系变量 S 中。

接下来，让我们来讨论一下更新和补偿性操作的情况。首先让我们考虑一下关系变量 S。假设我们向其中插入一个新供应商的数组，我们姑且设它为供应商 S6。为了保证普遍性，我们可以假设在关系变量 S 上的这个更新会与另一个在关系变量 SP 上的更新伴随出现，而在 SP 上的这个更新会向 SP 插入关于这个新供应商的一个设备供应数组集合。当然，如果实际上并没有设备供应的数组被插入，也只是说明这个新供应商的设备供应数组是个空集合。不管怎样，我们需要计算出这个设备供应数组中的数量之和，如此一来我们就可以向 STQ 中插入相对应的数组了（如果那个集合是空集，那么求和结果就是简单的零）。因此我们有如下的规则。

```
ON INSERT i INTO S :
   WITH ( t1 := SP MATCHING i ,
        ( t2 := SUMMARIZE t1 PER ( S { SNO } ) :
                                    { TQY := SUM ( QTY ) } ) :
   INSERT t2 INTO STQ ;
```

相对来说，在 S 上的删除规则就来得更加直接。

```
ON DELETE d FROM S :
   DELETE ( STQ MATCHING d ) FROM STQ ;
```

（当然，如果删除操作不额外级联到 SP 的话，那么该操作将会失败，同时在 SP 中

---

[8] 这个谓词与第 12 章中 SUMMARIZE 例子的谓词是一样的（但是约束是不同的，在后面的文章中我们很快可以看到）。不过严格来讲，第 12 章中的相关谓词应当被扩展为如下的样子：供应商 SNO“至少供应 1 个零部件并且”供应的零部件总数量为 TQY。

会有数组与被从 S 中删除的数组对应。如果删除操作确实级联到 SP 了，那么针对 SPd 的删除规则就会起效，不过该规则将没有任何效果。大家请参看下文。）

下面我们来看看在关系变量 SP 上的删除操作，其规则与第 12 章中对应的部分非常相似，估计与大家预想的类似，详情如下。

```
ON INSERT i INTO SP :
   WITH ( t1 := STQ MATCHING i ,
          t2 := SP MATCHING i { SNO } ,
          t3 := SUMMARIZE t2 PER ( S { SNO } ) :
                                   { TQY := SUM ( QTY ) } ) :
   DELETE t1 FROM STQ ,
   INSERT t3 INTO STQ ;

ON DELETE d FROM SP :
   WITH ( t1 := STQ MATCHING d ,
          t2 := SP MATCHING d { SNO } ,
          t3 := SUMMARIZE t2 PER ( S { SNO } ) :
                                   { TQY := SUM ( QTY ) } ) :
   DELETE t1 FROM STQ ,
   INSERT t3 INTO STQ ;
```

那么对于STQ上的更新又是怎样的呢？同样，删除规则很简单 [9]。

```
ON DELETE d FROM STQ :
   DELETE ( S MATCHING d ) FROM S ;
```

然而，通常来讲，插入操作是不合理的 [10]。例如向STQ中插入数组（S6，800）有什么意义呢？所以这里并没有插入规则。当然，没有插入规则并不意味着我们不能向STQ进行插入操作（比如，对SP的更新规则就会直接引起这类操作）。只是说这类操作很可能会因为违反黄金法则而失败，除非它们是某些多重赋值的一部分（而对于由SP的更新规则所引起的情况，它们确实是多重赋值的一部分）。

[9] 除了下面这种情况，如果删除操作不额外级联到 SP 的话，那么该操作将会失败，同时在 SP 中会有数组与被从 STQ 中删除的数组对应。

[10] 除了在第 12 章结尾时候的注释提到的情况之外。

第 14 章

# 通过表达式进行更新

当表达式相互冲突，视图相互矛盾，这些规则会预测将要出现的结果。

——Anon：《Where Bugs Go》

本书行文到这里，我首要关心的事情就是通过某些独立的关系操作进行更新的情况（更新一个限制、一个投影等）。除此之外，我一直半默认地假设，对于那些由比较复杂的表达式定义的关系变量来说，它们的更新规则可以由表达式中引入的各个操作的规则整合而来。举例来说，对一个由两个连接组成的合并进行更新，可以首先应用对合并的更新规则，然后再应用对连接的更新规则。我想现在是时候近距离观察一下这个假设了。

首先我要说的是这种假设从原则上来讲确实是正确的。原因是如果不这样做的话，那么结果实在是让人不忍直视！我们具体来体会一下，如果我们不这样假设，那就相当于把每一个表达式都当作一个特殊情况来处理。就上面这个例子来说也就是我们先要给两个连接的合并定义一组更新规则，然后再给两个不同的连接单独定义更新规则，而后再给由一个合并和一个差异组成的连接单独定义更新规则，如此这般周而复始无穷尽也。

我再强调一遍，这个假设是正确的，没有问题。然而，它确实也引出了特定的问题，而这些问题的答案却不见得总是完全清晰（至少目前对我来说是这样）的。因此，在本章中我有两个目标：第一，我想就这些问题进行解释和讨论；第二，我想试图经由这些讨论来给大家描绘出一些特定的结论。不过请一定注意，这一章不可避免地会带有一点推论或猜测的性质。坦率地讲，我承认目前我并不能给出所有的答案。

## 14.1 语义不是语法（?）

让我们来探讨以下示例（“示例 1”）。从大家都很熟悉的供应商关系变量 S 开始，假设我们定义视图 V1 和 V2 为如下形式。

```
VAR V1 VIRTUAL ( S WHERE CITY = ‘London’ OR CITY = ‘Paris’ ) ;

VAR V2 VIRTUAL ( ( S WHERE CITY = ‘London’ )
                   UNION
                 ( S WHERE CITY = ‘Paris’ ) ) ;
```

我们稍加观察就能看出这两个视图在语义上是等价的，但即使这样，它们的定义所使用的表达式从语法上来讲却是不同的。通过这个观察我想说的是在任意给定的时间，这两个表达式都表示相同的关系。因此，下面两点也很明显：（a）对于查询来说，如果 Q 是 V1 或 V2 中任一者的查询，那么它对另一个视图也同样适用，并且会生成相同的结果；（b）除此之外，对于更新来说也会有相似的特性。不过我想更仔细地看看（b）部分的特性是如何展现的。

首先，让我们先来看看 INSERT 操作。假设我们想要插入一个关于供应商 S9 的数组，该供应商所在城市为伦敦。那么：

- 在我们选用 V1 的情况下，通过限制进行插入的规则起效（大家可以参看第 4 章），最后的结果是新数组被插入到关系变量 S 中，并且也插入到 V1 中。
- 而如果选用 V2 的话，那么通过合并进行插入的规则会生效（大家可以参看第 10 章）。这个新数组满足对表达式为 S WHERE CITY = ‘London’这个具体合并操作数的限制条件，而不满足其他的合并，因此它会被插入到该操作数，而不是其他的。这时，通过限制进行插入的规则开始起效，于是最终结果就是新数组被插入到关系变量 S 中，并因而也插入到 V2 中。

其次，是 DELETE 操作。假设我们试图从中删除供应商 S1 的数组。那么有：

- 在选用 V1 的情况下，通过限制进行删除的规则生效（大家可以参看第 4 章），最后的结果是该数组被从关系变量 S 中删除，并且因而也被从 V1 中删除。
- 在选用 V2 的情况下，通过合并进行删除的规则生效（大家可以参看第 10 章 。该数组出现在表达式为 S WHERE CITY = ‘London’的这个合并操作数中，而不在其他的合并中，因此它被从该操作数中删除。这时，通过限制进行删除的规则开始生效，于是最终结果就是该数组被从关系变量 S 中删除，因而也被从 V2 中删除。

这样一来，我之前所声明的特性看起来是起效的，至少对这个例子来说是没有问题的。那么由此推想开去，它会不会引出这样一个具有吸引力的结论出现呢？通常来讲，如果视图V1 和V2 是如上面例子所描述的这样等价的，也就是说如果它们二者的定义表达式在任意给定时间都可以代表同一个关系，那么，如果U是V1 或V2 其中任一者的更新，就必然有U同样适用于另一者，而且具有完全相同的效果。实际上是会的，我已经就类似的问题发表过一系列略显教条的主张了 [1]，它们都与下面所写的内容一致。

**关于"语义不是语法原则"**：对于视图更新的语义不应局限于该视图定义表达式所使用过的那一条语句的语法。

注意：在我详细讲解这个"原则"之前，如果它确实是条原则，我需要澄清几点内容。首先，对于查询来讲，我之前所说的本质上是如果 Q 是 V1 的一个查询，那么同一个查询 Q 同样也对 V2 适用，并且会得出相同的结果。但是对于这两个视图来说，这个 Q 显然不可能是"完全一模一样"的，因为显然这两个视图的名字都不同。因此，举例来说，对于示例 1 给定如上面定义的视图 V1 和 V2，如果我说，下面这个查询

```
V WHERE STATUS > 15
```

是同时为 V1 和 V2 设计的，意思就是该查询中的符号 V 可以根据实际情况被替换为 V1 或 V2。这就是我之前说的表达式 Q 可以被认为是对两个视图的"同一个查询"。那么给定我们惯用的样本值，显然无论对于 V1，还是 V2，这个查询都会返回相同的结果。

| SNO | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |

以此类推，当我使用"同一个更新"这类词语的时候，我所指的也是类似的意思。

另外，我需要澄清的是关于我所谓的两个视图"等价"到底是什么意思。当然，在本章的讨论中我已经数次提到了对这个概念的直观理解，先是在我对示例 1 的研究中，后来又是在我要对该研究做一个宏观总结的时候。但现在的问题是：仅凭这个直观的理解是否足够支撑这个理论呢？我想大家已经猜到，这个问题的答案是"不

[1] 例如，"视图更新"就曾出现在我与 Hugh Darwen 合著的《Databases, Types, and the Relational Model: The Third Manifesto》第 3 版，Addison-Wesley，2006）一书的附录 E 中，以及其他文章和著作中。

能”。实际上，本章余下的大部分内容都将用来向大家更详细地说明视图等价到底是什么意思。

现在我们回过头来说“语义不是语法原则”。正如之前我说过的，我一直都假设这个“原则”是合理的。（不管怎样，它在另一种情境下是合理的。这里所说的情境，我在第 2 章中曾经做过解释，那就是补偿性操作并不由更新所使用的语句的语法所决定。）不过最近，我开始怀疑它到头来可能完全不成立，或者顶多是部分成立。我的态度会有如此转变的原因会在接下来的几节中向大家做出解释，但归根到底是这个原因，那就是太容易找到这个原则不能成立的例子了，至少是表面上不成立。除此之外，那些为了让这个原则成立所使用的小伎俩到头来也只是变得越来越巴洛克（不过是本轮外的本轮）。所有这些都使得下面这句Enrico Bombieri [2] 说过的话显得越来越正确（这也是我最喜欢格言之一）：“当事情变得过于复杂时，也许我们该停下来扪心自问：我问的问题是否正确？”请大家继续往下看……

## 14.2　一些广为人知的重言式（永真式、套套逻辑）

在逻辑学中，重言式（永真式、套套逻辑）指的是某些命题永远为真。更具体来说就是一个命题，对于每一次引用，无论其参数（如有）被替换为什么，都能保证该命题为真。举个例子，设 p 为随机命题，那么必有 p OR NOT p 为真，同样的例子还有 1+1=2（当然后面这一个其实是一个命题，只是它并不含有任何真正的参数）。

现在，形如exp1 ≡ exp的重言式（在这里exp1 与exp2 是任意表达式，而符号“≡”表示逻辑等价，即恒等）[3] 就显得尤为重要，因为它们允许一个包含exp1 的表达式被重写为一个包含exp2 的表达式。举例来说，就是令X1 为一个包含exp1 的表达式，exp1 为其子表达式，令exp2 逻辑等价于exp1，再将 X1 中的exp1 替换为exp2，令替换后的新表达式为X2。

为了更好地说明这个问题，我先带大家回顾一下 14.1 节中的示例 1。在该示例中，我实际上相当于声明了如下的表达式是一个重言式。

```
( S WHERE CITY = ‘London’ OR CITY = ‘Paris’ )
   ≡
( ( S WHERE CITY = ‘London’ ) UNION ( S WHERE CITY = ‘Paris’ ) )
```

[2] 菲尔兹奖获得者，于新泽西州普林斯顿高等研究院任“IBM 冯诺依曼”教授。

[3] 当且仅当 exp1 与 exp2 各自能够按照有效逻辑系统的推导规则，被对方推导出来时，我们说表达式 exp1 与 exp2 为逻辑等价。

当然，我确定大家对这些概念都很熟悉了，因为它们都源于关系优化过程中最基本的也是最重要的事情之一，即“表达式变换”，有时也被称为“查询重写”。大家请参阅《SQL and Relational Theory》来获取更多相关内容。

在集合论，尤其是关系代数当中，都有很多有趣的形如我们上面所讲的重言式即，exp1 ≡ exp 这种形式。下面有一些关系型示例与大家分享。令 A 和 B 代表同种关系类型下任意的关系表达式（在集合论中，它们应该代表任意集合）。于是我们有：

```
A ≡ A INTERSECT ( A UNION B )

A ≡ A UNION ( A INTERSECT B )

A INTERSECT B ≡ A MINUS ( A MINUS B )

A INTERSECT B ≡ B MINUS ( B MINUS A )

A MINUS B ≡ A MINUS ( A INTERSECT B )

A UNION B ≡ ( A MINUS B ) UNION ( A INTERSECT B ) UNION ( B MINUS A )

A MINUS A ≡ B MINUS B
```

接下来让我们看看上面这些概念对于视图更新到底有什么帮助吧。考虑下面这个例子（“示例 2”），我们引入在第 9、10、11 章中出现过的关系变量 PL 和 PK。在这里我要提醒你的是这两个关系变量都是只有单一属性 PNO（零部件号码），因此它们属于同一种关系类型，关系变量 PL 提供在售零部件的号码，而关系变量 PK 则提供有库存的零部件号码，所以二者的谓词都非常简单。

PL：零部件 PNO 正在销售。

PK：零部件 PNO 拥有库存。

样本值如图 14.1 所示（这里面的内容都是从第 9、10、11 章中的各个图表中摘录来的）。

PL

| PNO |
|---|
| P1 |
| P2 |
| P3 |
| P6 |

PK

| PNO |
|---|
| P2 |
| P5 |

图 14.1　关系变量 PL 和 PK 的样本值

接下来我们再来看两个视图，VPL1 和 VPL2，它们的定义表达式 exp1 和 exp2 如下所示（请大家回忆一下我们在第 3 章中介绍过的符号“≝”的意思是“定义为”）。

```
exp1 ≝ PL                                    /* defining view VPL1 */

exp2 ≝ PL INTERSECT ( PL UNION PK )          /* defining view VPL2 */
```

请大家注意观察，定义表达式 exp1 和 exp2 在这里是逻辑等价的（也就是说，exp1 ≡ exp2 是一个重言式，关于它的详细描述是在上面我给出的列表中的头一个）。因此，在任意给定时间，VPL1 和 VPL2 都拥有相同的值。实际上，它们所拥有的相同值与它们的底层关系变量 PL 的当前值是相等的。不过，让我们来看看下列删除操作。

```
DELETE ( P2 ) FROM VPL1 ;

DELETE ( P2 ) FROM VPL2 ;
```

当然，这两个删除操作对各自的目标视图本身都有着相同的效果。但是，正如大家都能很容易地发现，第 1 个删除操作将会导致 P2 数组被从关系变量 PL 中删除，但是对关系变量 PK 却没有任何影响。而对比来说，第 2 个删除操作同样会使得 P2 数组从关系变量 PL 中被删除，而同时也会使得该数组从关系变量 PK 中被删除，至少根据第 9、10 章中给出的通过交集与合并进行的删除规则是这样。于是这里显然就出现了在“等价的”视图上进行“同样的”更新却出现了不同效果的情况（虽然我之前说过它对视图本身的效果是一样的，但是至少是没有对底层关系变量起到相同效果）。

我们看到，对于一个能看到视图 VPL1 或 VPL2 二者之中任何一个，并且也可以看到底层关系变量 PL 以及 PK 的用户来说，上面说的删除操作在底层关系变量上的效果从道理上至少还说得通，当然我们假设该用户同时还知道相应的补偿性操作。但是总体来说这两个删除操作产生不同效果的事实依旧不变。那么显然我们的疑问也依然存在，那就是：到底为什么总体来说这两个删除操作会产生不同的效果？而这个现象又说明了什么呢？

在我尝试解答这些问题之前，大家先来看看下面两个在 VPL1 和 VPL2 上的插入操作。

```
INSERT ( P7 ) INTO VPL1 ;

INSERT ( P7 ) INTO VPL2 ;
```

同样地，这两个插入操作对各自的目标视图本身都有着相同的效果。很容易看出，第 1 个插入操作将会导致 P7 数组被插入到关系变量 PL 中，但是对关系变量 PK 却没有任何影响。第 2 个插入操作同样会使得 P7 数组插入到关系变量 PL 中被删除，

同时也会使得该数组插入到关系变量 PK 中，同样至少根据第 9、10 章中给出的通过交集与合并进行的删除规则是这样。于是我们再一次有了在“等价的”视图上进行“同样的”更新却出现了不同效果的情况，至少是对底层关系变量来说。

对于上述所有情况，我还要再雪上加霜一下，那就是在 VPL1 上不可能有更新操作与在 VPL2 上的上述删除或插入操作一样，能够对底层关系变量起到相同的效果，尤其是对底层关系变量 PK 而言。

那么显然，“语义不是语法原则”是有问题的。不过，在尝试解析它到底存在什么问题之前，我还要考虑另外一个相关的问题。

## 14.3 “语义变换”

在《SQL and Relational Theory》一书中，我简要地介绍了一种叫作“语义优化”的实施技巧。这里有一个简单的例子，是从早先那本书中摘录过来的。我们来看看这个表达式(SP JOIN S){PNO}。这里的连接是基于被引用的关系变量 SP 的外键，与被引用的关系变量 S 中对应的目标键之间的对应关系而来的。每一个 SP 数组都与某些 S 数组连接，也因此每一个 SP 数组都为产生整体结果的投影操作贡献了一个零部件号码。所以其实完全没有必要做连接！这个表达式可以被缩减到只具有赋值效果，如 SP{PNO}。不过请一定小心，这个变换仅仅在这个案例中的语义之下才适用。对于一般的连接来说，每一个操作数都会含有某些在其他操作数中没有对应关系的数组，它们并不向全局结果做贡献，也因此刚刚这种变换在这里就不适用了。但是就我们目前手中的例子来说，每一个 SP 数组在 S 中都必有一个对应项，因为刚好有一个外键约束的效果是每个设备供应都有一个供应商，所以最终变换是适用的。关于术语的解释：在特定约束起效的前提下适用的变换称为语义变换，由此达到的优化效果称为语义优化。

了解了上面的内容之后，让我们来看看下面的例子（“示例 3”）。我们简单一点假设关系变量 S 只有一个属性 SNO，而关系变量 SP 拥有两个属性 SNO 和 PNO（因此两个关系变量都是“全键”）。样本值如图 14.2 所示。

S

| SNO |
|---|
| S1 |
| S2 |
| S5 |

SP

| SNO | PNO |
|---|---|
| S1 | P1 |
| S1 | P2 |
| S1 | P3 |
| S2 | P2 |

图 14.2　关系变量 S 以及 SP 的样本值

下面我们考虑两个视图——VSP1 和 VSP2，它们的定义表达式分别为 exp1 和 exp2，详情如下。

```
exp1 ≝ SP                                  /* defining view VSP1 */

exp2 ≝ S JOIN SP                           /* defining view VSP2 */
```

请大家结合上面的讨论来观察这两个定义的表达式，其中exp2 可以“语义变换”为exp1[4]。由此我们可知，在任意给定时间，VSP1 和VPS2 都拥有相同的值，实际上，它们的值与关系变量SP在该时点的当前值一致。然而，让我们来看看下面的删除操作。

```
DELETE ( S2 , P2 ) FROM VSP1 ;

DELETE ( S2 , P2 ) FROM VSP2 ;
```

这里我们再次遇到了这种情况，这两个删除操作对各自的目标视图本身都有着相同的效果。我们很容易能够看出，第 1 个删除操作将会导致数组（S2，P2）被从关系变量SP中删除，但是对关系变量S却没有任何影响。而第 2 个删除操作同样会使得数组（S2，P2）从关系变量SP中被删除，而同时也会使得供应商S2 的数组从关系变量S中被删除，至少根据第 8 章中给出的通过一对多连接进行删除的规则是这样[5]。

接下来，我们再来看看下列插入操作。

```
INSERT ( S4 , P1 ) INTO VSP1 ;

INSERT ( S4 , P1 ) INTO VSP2 ;
```

第 1 条插入操作由于违反黄金法则而失败，更具体地说，是违反了从 SP 到 S 的外键约束。第 2 个插入操作则会使相关数组插入到关系变量 SP 中，同时也会使得供应商 S4 的数组插入到关系变量 S 中，至少根据第 8 章中给出的通过一对多连接进行插入的规则是这样。那么这一次，两个更新不仅仅是对底层关系变量的效果不同了，而且它们对视图本身的效果也不一样！换句话说，请大家注意这个重点，现在

[4] 或者通过别的方式实现，接下来的部分内容会支持这个主张。

[5] S 和 SP 的连接确实存在信息丢失，除非每个供应商都被要求必须至少供应 1 个零部件，因此你可能会认为罪魁祸首是通过这个连接进行更新所引入的编译指示。但其实并不是因为它。反而，这个问题看起来是一个固有问题。在这里我想简要介绍一下编译指示问题。我们已经看到了从视图 VSP1 和 VSP2 中删除数组（S2，P2）在关系变量 S 上会产生不同效果，这取决于哪一个视图是该操作的目标视图。不过至少它们对于视图本身都能够达到想要的效果。而反过来，某些评论家所拥护的观点是由于删除操作在像 VSP2 这样的视图上必须被完全拒绝（详见第 6 章），所以这会导致在 VSP1 上的删除操作成功，而在 VSP2 上失败！我认为，这种观点所阐述的情形引入了像第 6 章所描述的那样多的编译指示，并且如第 6 章所描述的那样“得体”。

我们遇到的情况是“相同的”更新甚至不能对它们自己的目标关系变量拥有相同的效果，尽管如此，这些目标关系变量却恰恰是“等价的”。

对于上面所说的问题，请允许我再次雪上加霜两条：（a）在 VSP1 上不可能有更新操作与在 VSP2 上的删除操作一样，能够对底层关系变量 S 起到相同的效果；（b）在 VSP1 上不可能有更新操作与在 VSP2 上的插入操作一样，能够对底层关系变量 S 或 VSP1 起到相同的效果。除此之外，所有 VSP2 上的更新都不会以 VSP1 插入操作那样的方式违反黄金法则。

很显然，这里发生的情况远比看到的要多。看来我们有必要近距离观察一下。

## 14.4 “信息等价”再回顾

让我们来简要地复习一下（同时也更加抽象一点）。

- 在示例 2 中，我们遇到了这样一种情况，视图的定义采用了 A INTERSECT (A UNION B)这种形式，涉及 B，但即使是这样，这个定义表达式仍然只与 A 逻辑等价。
- 在示例 3 中，我们遇到的情况是视图的定义采用了 A JOIN B 这种形式，同样也涉及到了 B，但即使是这样这个定义表达式在该示例中依然可以被“语义转化”为 A。

所以在这两条里最明显的一个问题就是：到底为什么 B 会被涉及到？

这些问题，以及和它们类似的问题非常令人烦恼。但是我想提醒你的是那句Enrico Bombieri说过的话：“当事情变得过于复杂时，也许我们该停下来扪心自问：我问的问题是否正确？”也许“到底为什么B会被涉及到？”并不是正确的问题。也许正确的问题应该是：说两个视图，或者说两个视图定义表达式，是等价的到底是什么意思[6]？

我想后面这个问题的关键在于对一个我在整本书中一直运用的概念的谨慎验证，即信息等价的概念。下面所引用的就是我在第 3 章中对这个概念的定义。

- **定义：**令 DB1 和 DB2 是若干关系变量的集合。那么 DB1 和 DB2 当且仅当适用于 DB1 和 DB2 的约束可以实现任何一个能够被 DB1 所表述的命题都可以被 DB2 所表述时，信息等价。反之亦然。

根据上述定义我们再来看示例 2，至少第一眼看起来应该能发现，只由 VPL1 组成

[6] 再次致谢 David McGoveran 帮我确定了本节的正确写作方向。

的设计与只由 VPL2 组成的设计确实是信息等价的，如此一来，在任意给定时间这两个关系变量的值都是相等的，而显然它们也代表着相同的命题。类似的，对于示例 3 来说，只由 VSP1 组成的设计与只由 VSP2 组成的设计应该也是信息等价的，这样的话在任意给定时间这两个关系变量的值都是相等的。但事实真地如此吗？

为了能够更深入地研究，让我们现在暂且将目标集中在示例 2 以及视图 VPL1 和 VPL2 上。下面我再次列出视图的定义表达式。

```
exp1 ≝ PL                                /* defining view VPL1 */

exp2 ≝ PL INTERSECT ( PL UNION PK )      /* defining view VPL2 */
```

那么我们能看到，确实exp1 和exp2 在这里是逻辑等价的（它们各自都能由对方逻辑推导出来）。但是它们并不是“信息”等价的！具体来说就是exp2 告诉了我们一些exp1 并没有告诉我们的事情，它告诉我们关系变量PK是存在的，而exp1 显然没有表明这一点 [7]。以此类推，现在我们考虑一下只由PNO的值P1 所组成的数组，它在两个关系变量中都会出现。而该数组在关系变量VPL1 中出现表达了下列命题。

零部件 P1 正在销售中。

而反过来看，该数组出现在关系变量 VPL2 中则代表了下列命题。

零部件 P1 正在销售中并且（零部件 P1 正在销售中或者零部件 P1 拥有库存）。

而正因为表达式 exp1 和 exp2 是逻辑等价，而不是信息等价，所以这两个命题也只是逻辑等价，而不是信息等价，也就是说，它们所传达的信息并不相同。最起码，第 2 个命题告诉我们零部件 P1 有可能拥有库存（这是一个很现实的信息，而第 1 个命题却压根没提到）。类似的，它还告诉我们零部件 P1 只有 3 种状态，要么在销售中，要么拥有库存，要么既在销售中，又拥有库存，这是第 1 个命题没有告诉我们的另一个很现实的信息。

有了前面的开胃菜铺垫，下面我要通过下列定义向大家展示一下何为“两个关系表达式是‘信息上的等价’”（不过现在我要把这个名字回归到我们常用的短语“信息等价”上来了）。

- **定义：**关系表达式 exp1 与 exp2 当且仅当下列条件满足时为信息等价。(a）它们是逻辑等价的，亦即它们各自都可以按照系统推导规则由对方推导出来，并

[7] 关系变量的名字其实可以很概念化地想成就是其谓词的一个缩写，除此之外再无其他。

且（b）在 exp1 中提到的每个关系变量也都在 exp2 中提到，反之亦然。

显然，对于视图 VPL1 和 VPL2 的定义表达式来说，虽然它们是逻辑等价的，但根据这个定义来看却不是信息等价的。那么，我们原本对于信息等价的定义，即对关系变量集合的那一个版本，讨论的是约束，而不是关系表达式，其中谈到两组关系变量是信息等价的当且仅当相应的约束起效：每个能被其中一组关系变量所表达的命题，同时也能被另一组表达。但是正如我们从本书前面部分数不清的讨论和案例中所得知的，在约束和关系表达式之间有一种一一对应的关系：对于每个关系表达式 rx，布尔表达式部分为 IS_EMPTY(rx)的约束都可以被定义；与此同时，每一个约束总可以通过这个 IS_EMPTY 的调用被公式化，因此对每个约束来说，这里都有一个相对应的关系表达式。特别是，我们还知道，每一个视图定义表达式都蕴含着一个特定的约束。我们可以肯定地说，当且仅当相应的视图定义表达式是信息等价时，两个这样的约束能够符合原始的定义要求，即信息等价那一个版本。

关于术语的说明：在示例 2 中，VPL1 和 VPL2 的定义表达式中所包含的关系变量名称 PL 和 PK，实际上被逻辑学家们称为“标志符”。当对包含它们的表达式求值时，它们的效果就是指示出一个值，也就是相应关系变量在当时的值。另外，请注意标志符与参数之间的逻辑区别。一个参数可以被任何一个主体取代，只要它是正确的类型。而相对而言，一个标志符根本就是不能，也无法被取代的；反而，它就像是编程语言中的变量，只能“指示出”或者表示相应变量在相应时点的值（也就是包含它的表达式需要求值的时候）。

现在，让我们把这些概念拓展到谓词和命题中去。VPL1 的谓词可以被象征性地，并且更正式一点地描述为以下的形式（这里我会用符号“⇔”来表示“当且仅当”）。

```
t ∈ VPL1 ⇔ t ∈ PL
```

（“当且仅当数组 t 出现在 PL 中时，它才会出现在 VPL1 中”。）类似的，VPL2 的谓词可以被描述为以下的形式。

```
t ∈ VPL2 ⇔ t ∈ PL AND ( t ∈ PL OR t ∈ PK )
```

（“当且仅当数组 t 出现在 PL 中，并且也出现在 PL 或 PK 二者之一之中时，它会出现在 VPL2 中”。）我做这些公式化修改的目的只是为了说明谓词也是可以被认为是包含了相应关系变量的。那么既然它们是这样的，我们也就可以对它们应用信息等价的概念了。当然，就目前的例子而言，我们可以看到这些谓词同样也不是信息等价的。除此之外，看起来下面这条叙述是合理的：如果谓词 P1 和 P2 是信息等价的，

那么对于每一个能从 P1 中实例化的命题，都有一个与之信息等价的命题可以从 P2 中实例化得到，反之亦然。

经过上面所有的这些分析讨论，我想现在我们有理由认为下面的情况是合理的。针对示例 2 而言，尽管表面上看起来并非如此，但最终还是出现了只由VPL1 组成的设计与只由VPL2 组成的设计信息不等价的情况。对于示例 3 也是类似的情况：VSP2 的定义中提到了关系变量S，而VSP1 的定义中则没有提到，因此只由VSP1 组成的设计与只由VSP2 组成的设计是信息不等价的。更具体点说就是VSP2 的定义告诉我们供应商关系变量是存在的，而在VSP1 的定义中却没有体现这一点 [8]。

现在我们回到示例 2。由上面的讨论可以得出结论，我们并不能说以下陈述是正确的，即在 T 时间对任意给定的在 VPL1 上的查询 Q 求值的结果，与在 T 时间对在 VPL2 上的相同查询 Q 求值的结果总是相等的。具体来说就是上面两个结果当然总是同一个关系 r，但是关系 r 中的数组集合代表的却是上述两种情况中不同的命题。因此下面这点，我更觉得完全合理：在 T 时间给定的在 VPL1 上的更新 U，与在相同时间 T 给定的在 VPL2 上的相同更新 U 有时会产生不同的效果。而我觉得下面的这一点简直更加合理，那就是在 VPL1 上的更新对关系变量 PK 从来都不会有任何效果，相反，在 VPL2 上的更新有时候会对关系变量 PK 有效果。我甚至觉得可以这样说，如果两个视图的定义表达式逻辑等价而并非信息等价的话，那么对这两个视图各自本身进行的相同更新有可能会产生不同的效果。事实上，我想大家一定记得这种情况恰恰在示例 3 中出现了。

最后我们来总结一下：对于“到底什么是两组关系变量的信息等价”这个问题的定义目前来看是成立的。而需要改进的是我们对下面这个问题的理解，那就是两个命题是“相同命题”这句话到底是什么意思。我已经阐述过当且仅当它们是信息等价的时候，它们才可以被认为是相同的，而对后面这个概念我也已经阐述了一个定义，那就是它们的谓词必须是信息等价的，而它们则是各自谓词的实例。

然而，这里依然有一个问题没有解决。我已经阐述过如果两个视图 V1 和 V2 的定义表达式并非信息等价的话，那么它们是不能互换的。但是这里依然有个问题：如果它们的定义表达式是信息等价的，那么它们就一定是可互换的吗？也就是说，即使 V1 和 V2 的定义表达式是信息等价的，那么是否有可能出现某个在 V1 和 V2 上相同的更新会产生不同的效果呢？我相信这个问题值得我们进一步研究。

[8] 对比来看，示例 1 就表现得很好，信息等价是成立的，因此没有类似的情况出现。

## 14.5　小结

我想给大家分享一些我的观察来结束本章，这些观察会展现这样的一种情况，那就是仅仅因为表达式 exp1 和 exp2 看起来在特定方面是等价的，并不能确保它们完全可以互换。不过请注意，这些内容可能会有点偏离本书的主题，如果大家不感兴趣可以跳过。

首先一点，我想说一个一般观点：那就是，我强烈怀疑我们目前遇到的问题，也就是通过"等价的"表达式进行更新的问题，是值与变量之间拥有显著逻辑差异的一种情形。更具体点说就是一个表达式和一个伪变量引用之间是有逻辑差异的。

一个表达式代表了值，而一个伪变量引用却不是。

（如果你需要复习一下关于伪变量的问题，你可以参看第 2 章结尾部分，而第 3 章可以帮你回忆起只要存在更新，视图的运行就像伪变量一样。）换句话说就是即使表达式与伪变量引用具有相同的语法，它们也拥有不同的语义。

接下来，我可以与大家分享过去的种种，彼时我仔细思考值与变量之间的逻辑区别并得到了顿悟（或者当时我确实觉得是顿悟，无所谓啦）。有一个例子是当时我和 Hugh Darwen 正在尝试开发的一种类型继承模型，它是"第三宣言"中类型理论的一个逻辑结果。我们当时正在纠结于"可替换性"这个概念，这是面向对象（object orientation，OO）世界中的一个概念，在面向对象的著作中对它通常是这样解释的：

如果S是T的一个子类，那么在系统中任何需要T类型的对象出现的地方，S类型的对象都可以替代它出现[9]。

例如，如果SQUARE（正方形）是RECTANGLE（长方形）的一个子类，那么按照上述方式用正方形替代长方形应该是可行的，因为正方形"也是"长方形。然而从表面来看，这个概念似乎会产生很多荒谬的结果，包括方形不是方的，圆形不是圆的，以及其他这类谬误（说实话，真是自相矛盾）[10]。后来当我们意识到面向对象的术语"对象"有时候指的是变量，而有时候指的是值时，我们才发现真正需要做的是将变量可替换性和值的可替换型区分开来，而正是这种区分工作让我们能够避免这些面向对象的谬误。更具体地说，正是这种区分工作令我们意识到对于特定表达式而言，即使它们可能在只读环境下完全自由交换，那也必须是在更新环境的前提下才能进行。

[9] 这个定义是从维基百科 Wikipedia (en.wikipedia.org)上摘录的。
[10] 顺便说一句，这个评价对 SQL 适用。

第 3 点我要说的还是与我们的继承模型有关，不过它可能与何谓两个表达式等价这个问题的意义关系更明显些。我在 14.2 节中提到的一件事是表达式A INTERSECT B与A MINUS (A MINUS B)是等价的，它们也确实是等价的，只要它们确实总是拥有相同值。然而，我们发现，首先，所有的表达式都具有“宣言”中所称的“声明类型”。其次，A INTERSECT B的声明类型总地来说与A MINUS (A MINUS B)的声明类型不一样！说得更具体点就是A MINUS (A MINUS B)的声明类型与A的一致，但是A INTERSECT B的声明类型却是A和B的“最小公共子类”，换句话说就是A类型的某种子类型。因此，我们并不能确保下面的命题成立，那就是：因为两个表达式总是具有相同值，所以它们就可以被互换使用。事实上，就我们手中的例子来说，虽然它们总是具有相同值，但它们甚至无法为了只读目的而互换！至少不是百分之百可互换[11]。

[11] 顺便说一句，这个情况说明，至少在类型继承相关环境下，对于两个关系表达式信息等价的定义可能需要增加一个条件，那就是相关表达式必须是相同声明类型。请大家参考 Hugh Darwen 与我合著的《Database Explorations: Essays on The Third Manifesto and Related Topics》（Trafford，2010）来获取更多信息。

# 第 15 章

# 歧义问题再回顾

让义愤的嘴巴闭上一会儿吧，直到我们能够扫清这里的歧义。

——威廉·莎士比亚《罗密欧与朱丽叶》（1592）

在前面几个章节中我有提过，David McGoveran 有一套方案来解决歧义问题并且可以在视图更新中加以发展（在没有信息等价的情况下）。在本书的最后一章，我想与大家讨论一下这套方案。不过在开始之前，有几点我一定要先讲清楚。

- 我们来讨论这个能解决歧义问题的方案并不是要对David的整套观点做整体评判。其实，他的整套观点是为了找到整体数据库遇到的更新问题而来的。它是由识别唯一真变量开始的，我在第 2 章也曾经解释过，从逻辑上讲，唯一真正能够被更新的就是整个数据库（或者叫数据库变量dbvar更好，不过我们并不使用这个名字）。数据库，以及其中的关系变量都是由谓词来描述的，不过这些谓词与我在本书中所说的那些并不相同，它们的本质是完全正式的。尤其是它们完全包含了我在第 2 章中所讲过的全局数据库约束。更新需求也是根据这样的谓词来表达（为了用户使用方便，它们可能会使用关系变量的名字，但是这类引用也仅仅是作为相应关系变量谓词的一个缩写）的。对一个更新的实施就包含了恰当地整合数据库谓词和更新谓词，以及计算能够符合这个整合谓词的一个或多个关系，这些关系就组成了整个数据库被更新的部分 [1]。如果大家想

[1] 这种方式产生的一个结果就是违反黄金法则的情况永远不会出现！如果 U 是一个通常总会违反某些约束的更新需求，那么 U 的整合谓词将会直接变为空集，它包含了所有相应的约束，因此没有任何更新操作会被真正执行。我们举例说明，看看这个显式UPDATE 需求，它试图将某些在关系变量NLS（“non London suppliers 非伦敦供应商”）中的数组的 CITY 值变为 London。这个 UPDATE 需求实际上是说 UPDATE NLS WHERE CITY = ‘London’，那么当然，在 NLS 中根本没有任何数组能够满足 WHERE 语句的要求。

了解更多细节，我建议大家去看看David的著作，我在第 3 章的“小结”中曾经引用过。

- 这里我还需要说明的是 David 可能压根就不会同意我把这些歧义问题叫作“歧义”这个名字。事实上，他相信有一种去歧义化的机制会一直伴随着我在本章中要讲的内容存在。（当然，如果确实有这么一种机制的话，那么可能就完全没有必要推广我在前面几章中提出的各种编译指示了。）换句话说，如果不使用这套去歧义机制的话，就相当于直接使更新需求变得不完整。在这种情况下，数据库管理系统 DBMS 很难判断哪种响应是最合适的（也就是说，所有可能的响应都变成了“正确的”）。

上面这些与我在本书中讨论的更新方式又有什么关系呢？首先，我相信这种方式是蕴含在 David 的方案精神中的，至少就我所理解的程度而言是这样。不过，我还是尝试用更加平实和通用的言语来讲解这种方式，我认为我所使用的这些言语在通常意义下的关系系统（也就是我在无数以前的著作和其他文章中所描述的关系系统）环境中都还是比较合理的。我也尝试过用这样一种方式来讲解这种方式，比如较为清晰地为这类关系系统提供一个可能的实施方案。

言归正传，现在让我来解释一下这个所谓的“去歧义机制”（或者叫“可能的去歧义机制”）到底是什么吧，它是在 David 著作中所描述的抽象概念基础上发展出来的一个解决歧义问题的具体方案。

## 15.1 谓词和约束再回顾

在开始之前，我想先来快速回顾一下在本书中所使用过的谓词和约束。在第 2 章我曾经说过，每一个关系变量都具有一个与之相关联的关系变量谓词，这个谓词简单来说就是这个关系变量对于用户来说到底是什么意思。举个例子，供应商关系变量 S 的谓词如下。

供应商 SNO 已经签约，名字为 SNAME，拥有状态 STATUS，并且位于城市 CITY。

为了表述简洁，我们把这个谓词简称为“pred（S）”，推而广之，让我们把任意给定关系变量R的谓词简称为“pred（R）”[2]。那么在任意给定时间T，在R中的每一个数组都应当是这样的：对于使用该数组的属性值替换“pred（R）”中的参数所产生的命题，具体说这个命题就是“pred（R）”在该时刻的一个具体实例，我们都相信它在T时刻为真。举例来说，对于（S1,Smith,20,London）出现在关系变量S中的

[2] 我采用这种标记，或者说命名方式，单纯地只是为了讲解目的。在实际操作中，我希望用户们能够根据需要选择他们自己认为“更友好”的谓词名称，而在后面的例子中我也会这样做。

这个事实，现在意味着我们相信下列命题在此时刻为真。

供应商 S1 已经签约，名字为 Smith，拥有状态 20，并且位于城市 London。

那么接下来，为了能够在各种可能的情况下正确地使用关系变量 R，用户必须知道（当然也得理解）这个谓词“pred（R）”。因此，关系变量 R 的“拥有者”或者“创建者”，或者也许是数据库管理员 DBA，必须用某种方式告诉所有会用到该关系变量的人到底这个谓词是什么。而且，这个做法对于无论是基础关系变量，还是虚拟关系变量（也就是视图），都同样适用。例如，在前面章节中出现过的对于视图 NLS（“non London suppliers”）的谓词“pred（NLS）”，详 情如下。

供应商 SNO 已经签约，名字为 SNAME，拥有过状态 STATUS，并且位于城市 CITY（CITY 不是 London）。

对于每个视图 NLS 的使用者来说，不管他的目的是什么，都必须知道并理解这个谓词。

正如我在其他著作中写到的（例如在《SQL and Relational Theory》中），在一个完美的世界中，谓词“pred（R）”应当是关系变量 R 的“更新操作可接受性标准”。但是目前来看这个目标显然是无法实现的。例如，对于关系变量 S 来说：

- 首先，数据库管理系统无法明白一个“供应商”已经“签约”或者“位于”某个地方是什么意思，这些都是“注释说明”的内容。举个例子，如果供应商号码 S1 和城市名称 London 刚好同时出现在同一个数组中，那么用户可以注释说明供应商 S1 位于 London，但是数据库管理系统却没办法做出类似的动作。

- 除此之外（也许这点更重要），即使数据库管理系统可以明白一个供应商已经签约或者位于某个地方是什么意思，但它依然无法判断用户所说的是真是假。如果用户（通过某些更新操作来）声称有一个供应商 S6 名字是 Lopez，状态为 30，并且位于城市 Madrid，那么数据库管理系统是无从判断该用户声称信息的正确性的，所有的数据库管理系统能做的只是检查这些内容有没有违反任何完整性约束。假设这些内容没有违反任何约束，那么数据库管理系统就会将数组（S6，Lopez，30，Madrid）插入到关系变量 S 中去，并且会从这个时点开始，将该数组所代表的内容作为一个真命题来对待。

换句话说，对于一个给定的关系变量来说，在今天的数据库管理系统中，实用的更新操作可接受性标准并不是关系变量谓词，而是相应的（全局）关系变量约束，这点我在第 2 章中也已经解释过。因此，有点一厢情愿地讲，对于给定的关系变量来

说，其关系变量约束应当作为其谓词的（近似）“数据库管理系统”存在[3]。当然，这个观点也解释了为什么我在本书前面的章节中如此强调约束的重要性。

## 15.2　一个交集示例

现在我们假设，有办法告诉数据库管理系统给定关系变量的名字和谓词，那么数据库管理系统应该最起码能够知道如“关系变量 S 的谓词是‘pred（S）’”这类信息。注意：我推测大概数据库管理系统会在目录中将这些谓词的名字与关系变量的定义保存在一起。我还推测大概也有办法能够将这些谓词的文本信息保存在目录中（为了在错误提示中显示，或者其他人机工程学目的），虽然我在这里讨论的这方面与视图更新本身没什么关系。当然，数据库管理系统还是不可能“理解”这些谓词到底是什么意思，但这不是问题的关键。

让我们继续假设，可以通过某种方式来要求数据库管理系统，使得我们可以用名字来引用这些谓词。例如考虑一下下面这个场景。假设我们有 2 个关系变量——A 和 B，它们的谓词分别为 pred（A）和 pred（B）。为了让这个例子看起来更实际一些，令 A 和 B 分别为第 9、10、11 以及 14 章中提到的关系变量 PL 和关系变量 PK。在这里我再次提醒大家，这些关系变量都只含有一个单一属性 PNO（零部件号码），关系变量 PL 提供在销售中的零部件号码，而关系变量 PK 则提供有库存的零部件号码。于是谓词 pred（PL）和 pred（PK）如下所示。

PL：零部件 PNO 正在销售中。

PK：零部件 PNO 拥有库存。

请大家注意这些关系变量都属于同一种关系类型。样本值如图 15.1 所示，同时该图中还分别包含相应交集、合并，以及差异关系变量——XLK、ULK，以及 DLK 的样本值（在下面我们还有更多讨论）。

PL

| PNO |
|---|
| P1 |
| P2 |
| P3 |
| P6 |

PK

| PNO |
|---|
| P2 |
| P5 |

XLK

| PNO |
|---|
| P2 |

ULK

| PNO |
|---|
| P1 |
| P2 |
| P3 |
| P5 |
| P6 |

DLK

| PNO |
|---|
| P1 |
| P3 |
| P6 |

图 15.1　关系变量 PL、PK、XLK、ULK 以及 DLK 的样本值

[3] 一厢情愿这话没错！很不幸，这个评价对现今的数据库管理系统还是太宽容了。一个令人沮丧的现实是现今大多数数据库管理系统（当然都是 SQL 系统）甚至都不能很好地支持约束，就更别说支持谓词了。

为了让我们接下来的讨论能够对用户更加友好一些，下面我要引入两个名字 ON_SALE 以及 IN_STOCK，用它们来分别替代看起来更正式一点的名字 pred（PL）以及 pred（PK）。（事实上这两个名字与我在第 9、10 以及 11 章中所使用的属性名称是相同的，在那些章中我用到这些名字的小节叫作"一个更好的设计"，而这并不是一个巧合。）下面，令关系变量 XLK、ULK，以及 DLK 都是视图，它们的定义分别是 PL INTERSECT PK、PL UNION PK，以及 PL MINUS PK（这些视图的样本值同样展示在图 15.1 中）。那么很明显，所有这些关系变量的谓词如下所示（"正式的"或者说符号化的版本如括号中所写）。

PL：零部件 PNO 正在销售中。（ON_SALE）

PK：零部件 PNO 拥有库存。（IN_STOCK）

XLK：零部件 PNO 既在销售中，又拥有库存。（ON_SALE AND IN_STOCK）

ULK：零部件 PNO 在销售中或者拥有库存。（ON_SALE OR IN_STOCK）

DLK：零部件 PNO 在销售中并且没有库存。（ON_SALE AND NOT IN_STOCK）

除此之外，既然我们假设数据库管理系统知道 ON_SALE 和 IN_STOCK 分别是关系变量 PL 和 PK 的"正式的"或者说符号化的谓词，那么它显然能够自己计算出关系变量 XLK、ULK，以及 DLK 各自相应的"正式的"或者符号化的谓词了。

正如我们从本书前面的章节中所知道的那样，如上面所述的这几个视图很可能会造成歧义问题。比如说，我们来看看在交集视图XLK上的一个更新，具体如下[4]。

```
DELETE ( XLK WHERE PNO = ‘P2’ ) FROM XLK ;
```

根据我在第 9 章中所描述的规则，这个删除操作的效果应当会导致零部件 P2 的数组被同时从 PL 和 PK 中删除。但是批评家们却表示反对，他们认为需求的效果（将数组从 XLK 中删除）可以经由单独从 PL、PK 或从两者中都删除该数组的方式达成，而这里却没有一个理由能够从这 3 种选项中选出一个最好的，因此这个删除操作应当被拒绝。

而我在第 9 章中所建议的是不管有没有歧义问题，都应当直接接受这个删除操作。当然，我在该章或者任何其他地方的观点并不是说我们对于这个问题放任不管，而是在我们能力所及范围内应当尽可能想办法解决歧义问题。那么让我们近距离观察一下。

[4] 在本章中我发现，比起前面几章用伪代码方式编写例子来说，使用更接近于纯 Tutorial D 语法的语句风格来编写例子是更加便利的。

假设一个用户U，他发起了上述针对关系变量XLK的删除操作。为了让问题尽可能简单，我们假设XLK是这个用户U所知道的唯一关系变量，也就是说，用户U根本不知道关系变量PL和PK的存在。尽管用户U不知道这些信息，但是他依然知道XLK的谓词是ON_SALE AND IN_STOCK这种形式，其中ON_SALE代表“零部件PNO正在销售中”，而IN_STOCK则代表“零部件PNO拥有库存”。因此如果用户U想要从XLK中删除零部件P2 的数组，那么就必然是这样的情况。即该用户是因为知道[5]下列 3 种可能性中哪一种成立，所以才想要进行删除。

1．零部件 P2 现在没有在销售（但是依然拥有库存）。

2．零部件 P2 现在没有库存（但是依然正在销售中）。

3．零部件 P2 现在既没有在销售中，也没有库存。

当然除此之外，用户 U 还必须知道到底哪一种情况是现在的实际情况，因为如果该用户不知道实际情况，那么从最初就没有理由更新数据库了！假设例子中现在遇到的是 2 号情况。那么（凭空编出点语句来）该用户可以并且应当发起下列删除操作（请注意**加粗**部分的内容）。

```
DELETE ( XLK WHERE PNO = ‘P2’ ) PRED ( IN_STOCK ) FROM XLK ;
```

采用“PRED (IN_STOCK)”这个写法的目的是让数据库管理系统知道，所要删除的数组应该都（只）满足谓词 IN_STOCK，“因此它们应当被（只）从关系变量 PK 中删除”。当然，在删除操作执行之后，这些数组将依旧存在于关系变量 PL 中，但是它们绝不会继续存在于交集视图 XLK 中了。

现在假设 3 号情况是我们实际遇到的情形，那么适用的PRED项就应当是PRED (IN_STOCK AND ON_SALE)[6]。而且看起来把这一项保留为默认项应该是很合理的一件事，这样的话如果没有显式PRED项存在的话，删除操作执行起来就会像第 9 章描述的一样了。

## 15.3　合并以及差异示例

我们再来回顾一下歧义问题至少是潜在有可能发生的几种情况，列举如下[7]。

[5] 其实用“相信”在这里可能比用“知道”要更好，但是我还是用“知道”来确保问题最简化。
[6] 这一项的作用当然是表示 XLK 的谓词，但是在下一节中我们会看到，这其实算是个偶然事件。
[7] 这里我说“至少是潜在”的意思是说某些情形下已经丧失了信息等价。大家可以特别注意观察一下第 14 章中所描述的那些各种各样反常的例子。

1．通过一个交集来进行删除操作（请参看第 9 章）。

2．通过一个合并来进行插入操作（请参看第 10 章）。

3．通过一个差异来进行删除操作（请参看第 11 章）。

当然，在 15.2 节中我们已经讨论过第 1 种情况了。只不过现在我要指出，交集是连接的一种特殊情况，因此在 15.2 节中所展示的观点我们当然可以推而广之地应用到连接视图中去。不过现在还是让我们来看看另外两种情况吧。

## 15.3.1 合并

首先我们来说合并。我们来看看下面这个在合并视图 ULK 上的更新操作。

```
INSERT RELATION { TUPLE { PNO ‘P4’ } } INTO ULK ;
```

（表达式 RELATION {TUPLE {PNO ‘P4’}}是一个关系选择器调用，表示该关系只包含这个具体的数组。）那么，根据第 10 章中所描述的规则，这个插入操作的效果应当是令该数组同时被插入到 PL 和 PK 中。其实该需求所要求的效果，向 ULK 中插入该数组，可以通过将该数组插入到 PL 或 PK 二者之一实现，而并没有必要同时插入到二者之中。

但是我们要从用户角度考虑一下，假设还是用户 U 发起了上面这个插入操作。为了让问题最简化我们假设 ULK 是用户 U 所知道的唯一关系变量，也就是说，用户 U 根本不知道关系变量 PL 和 PK 的存在。尽管不知道这些信息，但是用户 U 依然知道 ULK 的谓词是 ON_SALE OR IN_STOCK 这种形式（同样，其中 ON_SALE 代表“零部件 PNO 正在销售中”，而 IN_STOCK 则代表“零部件 PNO 拥有库存”）。因此如果用户 U 想要向 ULK 中插入零部件 P4 的数组，那么就必然是这样的情况：该用户是因为知道下列 3 种可能性中哪一种成立的，所以才想要进行插入。

1．零部件 P4 现在正在销售中（但是没有库存）。

2．零部件 P4 现在拥有库存（但是没在销售中）。

3．零部件 P4 现在既在销售中，也有库存。

当然除此之外，用户 U 还必须知道到底哪一种情况是现在的实际情况。假设例子中现在遇到的是 2 号情况，那么该用户可以并且应当发起下列插入操作。

```
INSERT RELATION { TUPLE { PNO ‘P4’ } } PRED ( IN_STOCK ) INTO ULK ;
```

这样新数组就会（只）被插入到关系变量 PK 中了。

现在假设 3 号情况是我们实际遇到的情形，那么适用的PRED项就应当是PRED (IN_STOCK AND ON_SALE)[8]。看起来把这一项保留为默认项应该是很合理的一件事，这样的话如果没有显式PRED项存在的话，插入操作执行起来就会像第 10 章描述的一样了。

### 15.3.2 差异

现在该来说说差异了。我们来看看下面这个在差异视图 DLK 上的更新操作。

```
DELETE ( DLK WHERE PNO = ‘P1’ ) FROM DLK ;
```

根据第 11 章所描述的规则，这个删除操作的效果应当是令该数组被从 PL 中删除并被插入到 PK 中。其实该需求所要求的效果，是从 DLK 中删除该数组，这只需要通过上述两个操作之中任意一个即可实现，没有必要将两个操作都执行。

但是我们还是要从用户角度考虑一下，假设还是用户 U 发起了上面这个删除操作。为了让问题最简化我们假设 DLK 是用户 U 所知道的唯一关系变量，也就是说，用户 U 根本不知道关系变量 PL 和 PK 的存在。尽管不知道这些信息，但是用户 U 依然知道 DLK 的谓词是 ON_SALE AND NOT IN_STOCK 这种形式（同样，其中 ON_SALE 代表“零部件 PNO 正在销售中”，而 IN_STOCK 则代表“零部件 PNO 拥有库存”）。因此如果用户 U 想要从 DLK 中删除零部件 P1 的数组，那么就必然是这样的情况：该用户是因为知道下列 3 种可能性中哪一种成立，所以才想要进行删除的。

1．零部件 P1 现在没有在销售（而且也没有库存）。

2．零部件 P1 现在拥有库存（而且也在销售中）。

3．零部件 P4 现在没有在销售，但是拥有库存。

假设例子中现在遇到的是 2 号情况，那么该用户可以并且应当发起下列删除操作。

```
DELETE ( DLK WHERE PNO = ‘P1’ ) PRED ( IN_STOCK ) FROM DLK ;
```

在这里的 PRED 项应当这样理解：该零部件 P1 的数组现在满足关系变量 PK 的谓词（当然它之前不满足，否则它就不会出现在关系变量 DLK 中了），因此它需要被插入到关系变量 PK 中，同时不被从关系变量 PL 中删除。注意：在这里我做个记录，适用于 1 号情况的项应当为 PRED (NOT ON_SALE)，它可以使该数组被从关

---

[8] 大家注意观察，AND 在这里是正确的，我们不应当使用 OR，即使 ULK 的谓词是 IN_STOCK OR ON_SALE。

系变量 PL 中删除，同时不被插入到关系变量 PK 中去。而适用于 3 号情况的项应当为 PRED (NOT ON_SALE AND IN_STOCK)，它可以使该数组被从关系变量 PL 中删除，同时被插入到关系变量 PK 中去。看起来把这一项保留为默认项应该是很合理的一件事，这样的话如果没有显式 PRED 项存在的话，删除操作执行起来就会像第 11 章描述的一样了。

## 15.4　更多关于谓词的内容

一个早期版本的审阅者曾经对本章内容表达过这样的疑虑，那就是前面两节我所提出的观点是否真地有效。具体来说，他（该审阅者是位男性）提出，假如一个视图被定义为 A UNION B，那么用户不一定会知道其谓词是 pred(A) OR pred(B)。这种模式。之后他进而给出了一个例子来加以佐证，这里有 2 个关系变量 MOTHER 和 FATHER，它们各自都只有一个属性叫作 NAME，而谓词则分别是“NAME 是某人的母亲”，以及“NAME 是某人的父亲”，那么 MOTHER UNION FATHER 的谓词可能就不是“NAME 是某人的母亲”OR“NAME 是某人的父亲”，而是简单的“NAME 是某人的家长”。

为了回应这个例子，让我来研究一下，首先关系变量 MOTHER 和 FATHER 显然是交集为空的，那么它们的合并就应该是一个交集为空的合并。因此该合并的谓词并不完全是我在前面几段中给出的样子，而应该是如下这样。

（“NAME 是某人的母亲”OR“NAME 是某人的父亲”）

AND

NOT（“NAME 是某人的母亲”AND“NAME 是某人的父亲”）

因此，如果这个合并视图的用户真的只知道谓词“NAME 是某人的家长”，那么该用户就不可能知道 INSERT 操作在这个视图上必须将目标指向 MOTHER 和 FATHER 这两个关系变量其中之一。由此，这种操作将会直接失败，因为它违反了黄金法则。

如此说来，不管是数据库管理员，还是其他什么人，负责告知该用户这个视图谓词的相关内容，如果那个人决定不去告诉用户正确的谓词是什么，那么我将不得不质疑他的判断力。但是你没法用法律限制判断力，对吗？

说到这里，如果合并视图并不代表一个交集为空的合并，那又会是怎样的呢？让我们来看一个现实的例子（也是第 10 章中我们简短讨论过的一个例子的变种）。假设某些零部件是在国外生产的，另一些则是在国内生产的，而还有一些则是国内和国

外都生产的。那么我们进一步假设使用两个关系变量来代表这种情况，二者都只有一个单一属性 PNO，其中一个（PA）提供在国外生产的零部件号码，另一个（PD）则提供国产的零部件号码。那么这两个关系变量的合并，我们叫它 UAD，就是一个重叠合并，而它将提供所有零部件的零部件号码。各个关系变量的谓词如下所示。

PA：零部件 PNO 在企业内使用并且在国外生产。

PD：零部件 PNO 在企业内使用并且在国内生产。

UAD：零部件 PNO 在企业内使用并且不是在国外生产的，就是在国内生产的（也有可能国内国外都有生产）。

如果用户被告知 UAD 的谓词只是“零部件 PNO 在企业内使用”，那么插入操作就必须像在第 10 章中描述的那样进行，故事结束。然而，不论是谁告诉用户这就是该合并的谓词，我都要再次质疑他的判断力。（另外，请大家注意这里与第 14 章“信息等价再回顾”一节中讨论内容的相关性。）

## 15.5 小结

我想通过再次重复和强调在本章开头我所提过的一点（我提到歧义问题只有“当信息等价丢失时”才会产生）来结束本章，以及本书的正文部分。确实，各种各样的歧义问题会伴随着前面章节所讲述的视图更新方式出现，而各种各样对该方式的反对意见也已经被特定的批评家们提出，它们都具有同一个根本原因。那就是，有的时候，数据库管理系统所获得到的信息不够充足。例如，假设数据库含有两个含有相同关系头的不同关系变量，就像在本章和前面章节中各种示例引入关系变量 PL 和 PK 一样。显然这些关系变量代表着不同的信息（因为如果信息相同，就没必要两者都要了），换句话说，它们对应着不同的谓词。但是如果这些谓词因为某些原因不被数据库管理系统所知道，例如它们存在于系统之外，比如只在某些用户的脑海中，那么数据库管理系统就不可能知道怎样做是正确的了。

那么我个人所推荐的对这些问题的解决办法，都在第 9～11 章中各自的“一个更好的设计”这一节之内讨论过了。在这里我要提醒大家，对于关系变量 PL 和 PK 来说，这个解决方案涉及到了对两个属性 ON_SALE 和 IN_STOCK 的引入（它们都是 BOOLEAN 类型，并且都有很明显的解释）。事实上，我所做的只是重新设计了数据库，从而可以给数据库管理系统提供其他的隐藏信息，特别是重新设计数据库，从而使数据库管理系统可以自己判断给定的数组逻辑上属于哪一个关系

变量。（我们换一种方式来说，就是引入这些属性使得相应的谓词可以直接由约束代表，这样的效果相当于让这些谓词“被数据库管理系统知道了解”。）但是，如果有任何原因导致设计者认为不用这种合理的方式设计数据库也挺好，那么我想唯一能让数据库管理系统总能做出正确选择的办法，就是我在本章中所表述的这种方法了。

# 附录 A

# 关于关系赋值的一点意见

改变都会伴随着不便

——Samuel Johnson《A Dictionary of the English Language》（1755）

在第 2 章中我曾经说过，一般关系赋值：

```
R := rx
```

（其中R是个关系引用，或者换种说法，从语法上讲R是一个关系变量的名字，而rx是一个关系表达式）逻辑上等价于下列形式的赋值[1]。

```
R := r - d ∪ i
```

实际上，在第 2 章中我使用了关键词 MINUS 和 UNION 来分别代替集合的差异符号“-”以及集合的合并符号“∪”。在本篇附录中我发现符号用起来更方便些。其他我还会使用到的符号包括“∩”（集合相交符号）、“⊆”（集合包含符号）以及“Ø”（空集符号）。除此之外，在整个篇幅中我所说到的应该都是关系，而不是普通集合，不过我建议大家在后面不用在意这个细节。注意：在第 2 章中我曾经用圆括号括住了一个子表达式 r−d（或者说是 r MINUS d），不过其实并没有必要加上那个括号，在接下来的文章中我们会看到解释。

那么接下来，就让我来详细介绍一下我的概念，并且对一些重点内容做出详细说明。

- r 是赋值之前 R 的值（也就是 R 的“旧”值），d 是要被从 R 中删除的数组的集合（“删除集合”），而 i 是要被插入到 R 中的数组集合（“插入集合”）。注意：

[1] 在本附录中，我将使用非标准化的术语“赋值”来专门指代关系赋值。

更具体地说，d 和 i，与 r 相似，确实是关系，而我所说的这些数组集合其实都是关系体。不过后面我也不会再强调这些细节了。

- d⊆r（与 d−r = Ø 等价），因为删除一个根本不存在的数组首先从逻辑上来讲就是没有效果的。换句话说就是我们没有必要把 d 的范围变得比所需要的更大。
- r 和 i 交集为空，即 r∩i = Ø，因为插入一个本就存在的数组从逻辑上讲是没有效果的。换句话说就是我们没有必要让 i 的范围变得比所需要的更大。
- d 和 i 交集为空，即 d∩i = Ø，因为在同一个赋值操作中先删除一个数组，然后再把它插入进来（或者其他类似方式的操作）是没有意义的。当然，这一点 d∩i = Ø 是前面两点 d⊆r 和 r∩i = Ø 的逻辑结果。

这些要点可以用一张文氏图来表现（请参看图 A.1，与第 2 章中的图 2.2 相同）。解释说明：在该图中，r、d，以及 i 都是上面所描述的集合，而 u 则是相应关系类型的全集（也就是关系体有所有与 R 具有相同关系头的关系，当然其中包含那些当前出现在 R 中的数组）。请注意差异 u−r（显然）是 r 的补集，即那些关系体有所有与 R 具有相同关系头的关系，除去当前出现在 R 中的那些。

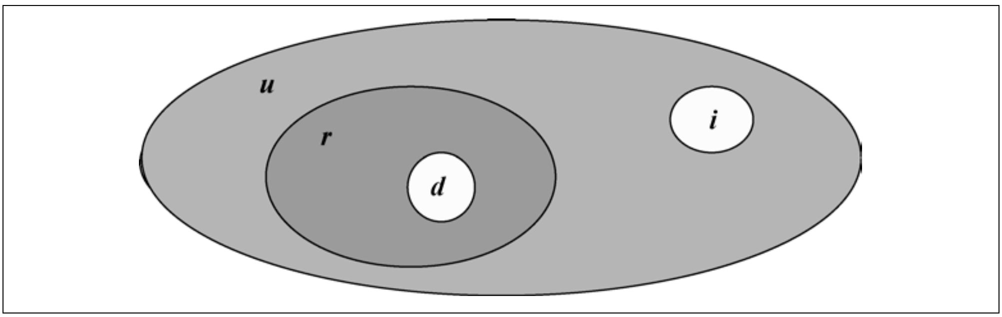



图 A.1　删除集合 d 以及插入集合 i

注意：当然，在赋值操作本来就是个 INSERT 操作的情况下，删除集合 d 可以是空集。在赋值操作本来是 DELETE 操作的情况下，集合 i 可以是空集。而如果赋值操作降级为一个空操作“no op”的话，那么这两个集合则可以都为空。

下面我们来看这个例子，给定赋值操作 R: = r − d∪i（其中 r、d 以及 i 都是前面文章定义的集合），删除集合和插入集合都是唯一的。那么对于 d1≠d2 和/或 i1≠i2 的情况，现在假设：

```
r - d1 ∪ i1 = r - d2 ∪ i2
```

其中 d1−r = d2 − r = r∩i1 = r∩i2 = d1∩i1 = d2∩i2 = Ø。那么有：

- 令 $t \in d1 - d2$。因为 $d1 - r = Ø$、$t - r - d2$ 且 $t \notin r - d1$，所以 $t - r - d2 \cup i2$，$t - r - d1 \cup i1$。因为 $t \notin r - d1, t - i1$，但又因为与 $d1 \cap i1 \neq Ø$ 矛盾，所以 t 不存在，且 $d1 - d2 = Ø$，即 $d1 \subseteq d2$。类似地可以得出 $d2 \subseteq d1$，所以 $d1 = d2$。

- 于是我们设 $r - d1 = r - d2 = z$。那么 $z \cup i1 = z \cup i2$，但是 $r \cap i1 = Ø$，所以 $(r - d1) \cap i1 = Ø$，$z \cap i1 = Ø$。类似地可以得出 $z \cap i2 = Ø$，由于 $z \cup i1 = z \cup i2$，那么有 $i1 = i2$。

从上面我们可以得出如果关系 x 被赋值给 R，那么 x 可以被表达式 $r - d \cup i$ 或者 $r \cup i - d$ 所表示，即我们是“先做删除”，还是“先做插入”，是没有区别的。我们还能由此得出，给定 r 和 x，我们有：

```
d = r - x
i = x - r
```

现在，我来重复一下之前我说的一点，INSERT 基本上就是 d 为空集时的赋值操作，而 DELETE 基本就是 i 为空集时的赋值操作。不过接下来我想说说关于 UPDATE 这个操作的事情（当然是对显式 UPDATE 操作而言）。显然，对这个操作来说，一般情况下 d 和 i 必须都为非空集合。那么不如让我们来看看一般格式下的这类操作。

```
UPDATE R WHERE bx : { A := ax }
```

在这里 R 是一个关系变量引用，bx 是一个布尔表达式（默认为真），A 是关系变量 R 的某个属性，而 ax 是一个表达式表示与 A 相同类型的值 a。（为了表述简洁，我只关注涉及到单一“属性赋值”的 UPDATEA: = ax。而接下来的讨论则会将这个 UPDATE 推而广之到涉及两个或更多赋值的情况。）

给定这个 UPDATE，很明显删除集合 d 的定义应该如下所示（这里要提醒大家这个符号“≝”的意思是“定义为”）。

```
d ≝ { t : t ∈ R AND bx ( t ) AND t.A ≠ ax }
```

解释说明：这个定义的意思是 d 仅由满足下列条件的数组 t 组成。首先，t 在 R 中出现；其次，bx 对 t 求值为真；第三，t 中的 A 值已经不等于 a 了。

插入集合 i 由 d 定义，于是有：

```
i ≝ { t′ : EXISTS t ∈ d ( t′ = EXTEND t : { A := ax } ) }
```

解释说明：在本书中我们所见到的关键词 EXISTS 表示的是存在量词（它可以被读

作“这里存在…”)。那么，这个定义的意思就是 i 仅由满足下列条件的数组 t'组成，即在 d 中存在数组 t 与这个数组 t'相等，除了数组 t 中的 A 值，在数组 t'中已经被值 a 所替代。注意：这里我所使用的是“假设情况”版本的 EXTEND（大家可以参看第 2 章或者附录 B）。也许更重要的是我同时也应用了（其实在文中其他很多地方也有提到）这样一个事实，那就是我们可以定义许多常规的关系型操作，特别是包括 EXTEND，从而可以对某些特定数组，而不是整个关系应用。关于这一点请大家参看《SQL and Relational Theory》来获得更多资讯。

由上面的所有内容我们可以知道，一般情况下显式的 UPDATE 操作在 d 和 i 都是非空集合的时候确实相当于一个关系赋值。然而一定要指出的是反过来却是不对的。也就是说，当 d 和 i 都是非空集合的时候，并不是所有的关系赋值都等同于某些显式 UPDATE 操作。实际上，原因也是显而易见的，罗列如下。

- 简单来说，我们可能会遇到一个显式 UPDATE 操作反应的是数组 d 和数组 i 之间一一对应的关系的情况。更具体点来说就是：首先，d 中的每一个数组都对应 i 中的一个数组（不过请注意，d 中的这两个或多个不相同的数组可能对应的是 i 中的同一个数组，但是这种情况仅当这个 UPDATE 是一个“键 UPDATE”的时候才可能发生，而这种情况也是极为罕见的)。其次，i 中的每一个数组都至少与 d 中的一个数组对应（实际上，应该是 d 中唯一的一个数组，不过也是当 UPDATE 恰好是“键 UPDATE”时例外)。
- 在一般意义下的关系赋值中，这样的特性不成立。例如，一个具体的赋值可能要删除 3 个数组并插入 5 个数组。

我在本附录中要讨论的最后一个问题是针对重复目标进行多重赋值的情况。这里我还要再次说明多重赋值的语义，这个我在第 2 章中曾经讲过。

我们概括来讲，多重赋值的语义有下面 2 条。

- 首先，在等式右侧的独立赋值的源表达式进行计算。
- 然后，那些独立赋值操作被执行。

我们来观察一下，正是因为所有的源表达式都要先进行计算，然后独立赋值再执行，所以那些独立赋值都不会被其他运算结果影响，也因此它们被执行的顺序也跟运算结果无关（如果愿意，你甚至可以把它们的执行想象成平行发生的，或“同时进行”的)。

不过，我也提到过上面的解释需要一点点改进，以便更好地应对两个或多个独立赋

值的目标变量相同的情况。下面我将简要介绍一下这个改进。

首先，我们来看看下列的简单例子。

```
DELETE ( S WHERE SNO = ‘S1’ ) FROM S ,
DELETE ( S WHERE SNO = ‘S2’ ) FROM S ;
```

而下面是相应扩展后的形式。

```
S := S MINUS ( S WHERE SNO = ‘S1’ ) ,
S := S MINUS ( S WHERE SNO = ‘S2’ ) ;
```

很显然，如果两个源表达式在某个单独赋值执行之前进行了求值运算，那么其中一个赋值就会没有效果！换句话说，就是 DELETE 之一“丢失了”。（而更严重的是丢失的那个赋值显然都还没有被定义过。）

正因为上面所说的这种情况，所以多重赋值的定义扩展了一部分，那就是如果该多重赋值中两个或多个独立赋值对应的是同一个赋值目标变量，那么这些独立赋值会按照编写的先后顺序来执行。更准确点说就是上面的双重赋值逻辑上等价于下列的单一赋值。

```
S := WITH ( S := S MINUS ( S WHERE SNO = ‘S1’ ) ) :
                                   S MINUS ( S WHERE SNO = ‘S2’ ) ;
```

而这个赋值又与下面的赋值等价。

```
WITH ( DELETE ( S WHERE SNO = ‘S1’ ) FROM S ) :
       DELETE ( S WHERE SNO = ‘S2’ ) FROM S ;
```

当然，这里我们所讨论的双重赋值逻辑上也等价于下列单一赋值。

```
S := ( S MINUS ( S WHERE SNO = ‘S1’ ) ) MINUS ( S WHERE SNO =
‘S2’ ) ;
```

更概括地讲，很明显，一个所有内在的独立赋值都瞄准同一个目标关系变量 R 的多重赋值，逻辑上等价于针对该关系变量 R 的某个单一赋值。为什么？因为任何赋值，无论多重与否，都相当于对 R 赋了一个“新”值 R - d∪i，其中 d 和 i 为某集合。

当然，以上的研究虽然是正确的，但是对通常情况下计算 d 和 i 的值并没有什么帮助。不过下面的内容可能对这个问题还真有些帮助。我们来看看下面的双重赋值（并且请一定注意，在第 2 个赋值右侧表达式中的关系变量引用 R，表示的是第 1 个赋值执行之后关系变量 R 的值。

```
R := R - d1 ∪ i1 , R := R - d2 ∪ i2 ;
```

那么，一般情况下，下面这几个集合对并不一定要是交集为空的：d1 和 d2、i1 和 i2、d1 和 i2、d2 和 i1。不过，我们很容易看出（这个留给读者做练习）上述双重赋值逻辑等价于下面这个形式的单一赋值。

```
R := r - d ∪ i ;
```

其中：

- r 是 R 的初始值，即 R 在第 1 个赋值执行之前的值。
- $d = ( d1 - i2 ) \cup d2$。
- $i = ( i1 \cup i2 ) - d2$。

注意这里的 d 和 i 一定是交集为空的，不过通常来讲并非如此，一般来说 d 包含在 r 中或者 i 与 r 交集为空。下面我们定义：

- $d' = ( r \cap d ) - i$
- $i' = i - ( r \cup d )$

那么原来的双重赋值就逻辑等价于下面的单一赋值。

```
R := r - d′ ∪ i′ ;
```

其中 $d' - r = r \cap i' = d' \cap i' = \varnothing$。

那么注意，综上所述，最终我们会得到对于如下的多重赋值：

```
R := rx1 , R := rx2 , ... , R := rxn
```

来讲，对任意 $n > 0$ 都可以降级为一个单一赋值。

```
R := r – d ∪ i
```

其中 r 为 R 的初始值，并且有 $d - r = r \cap i = d \cap i = \varnothing$。

附录 B

# 关系运算符

文明的进步，就是不断增加我们可以不加思索就能执行的重要活动的数量

——Alfred North Whitehead：《Introduction to Mathematics》（1911）

在本附录中，我将为所有在本书中讨论过的关系运算符给出相关定义，以便大家查阅。所有的定义都将按照名称的字母顺序排序给出（以在 Tutorial D 中的操作名称为准），这些定义都是基于《The Relational Database Dictionary, Extended Edition》（Apress，2008 年）书中的定义而来的，只是针对本书做了一些微小的简化而已。请特别注意观察，所有的运算符都会返回一个定义好的关系头，以及一个定义好的关系类型，因而任何一个调用该运算符的表达式都属于这个关系类型。关于这个问题的更多内容，请大家参看《SQL and Relational Theory》。

- **EXTEND（展开）**：1. （新属性）令关系 r 并没有一个属性叫作 A。那么表达式 EXTEND r : {A := exp}会返回这样一个关系，它具有 r 的关系头并由属性 A 展开，而关系体则是由数组 t 组成的，其中 t 是 r 被 A 的值扩展的数组，该 A 的值则是针对 exp 在该数组上进行求值得到的结果。2. （已存在属性）令关系 r 拥有属性 A。那么表达式 EXTEND r : {A := exp}将返回这样一个关系，它具有 r 的关系头，而关系体则是由数组 t 组成的，其中 t 是这样得出的：首先对 exp 在该数组上进行求值得出值 a，然后用 a 替代 A 的值并导出 t。

- **GROUP（组合）**：令 r 的关系头被分为子集 X = {X1, X2, ..., Xm}和 Y = {Y1, Y2 ...,Yn}，同时，用 YR 命名一个属性名称，该属性不出现在 X 中。那么表达式 r GROUP ({Y1, Y2, ..., Yn} AS YR)将会返回一个关系 s。这个关系 s 的关系头为{X1, X2, ..., Xm, YR}，其中 YR 的类型为 RELATION {Y1, Y2, ..., Yn}。

关系 s 的关系体则被定义为：令 z 为 r WRAP ({Y1, Y2, ..., Yn} AS YT)的结果。对于 z 中每一个不同的 X 值 x，令 yr 为一个关系，其中的数组包含全部 z 中数组的 YT 值并且仅包含 YT 值，令数组 t 的关系类型为 TUPLE {X, YR}，并包含 X 的值 x 以及 YR 的值 yr，那么只有在这种情况下，t 为 s 的数组。

- **INTERSECT（交集）**：1. （二元情况下）令关系 r1 和 r2 拥有相同的关系头 H。那么表达式 r1 INTERSECT r2 将会返回一个关系，其关系头为 H，关系体由数组 t 组成，其中 t 同时出现在 r1 和 r2 中。2. （多元情况下）令关系 r1，r2，…，rn（n ⩾ 0）都拥有相同的关系头 H。那么表达式 INTERSECT {r1, r2, ..., rn}将会返回一个关系，其关系头为 H，关系体均由数组 t 组成，其中 t 同时在 r1，r2，…，rn 中出现。注意：如果 n=0，那么会有（a）虽然在这里没有展示出来，但是需要在相应的关系头中详细指定某些语义的功能，（b）结果是具有相应关系类型的全集关系。

- **JOIN（连接）**：1.（二元情况下）令关系 r1 和 r2 拥有名称与类型完全相同的属性。那么表达式 r1 JOIN r2 将会返回一个关系，其关系头为 r1 与 r2 关系头的并集，而关系体由数组 t 组成，其中 t 是 r1 中的一个数组与 r2 中的一个数组的并集。2.（多元情况下）令关系 r1，r2，…，…rn（n ⩾ 0）拥有相同名字以及相同类型的属性。那么表达式 JOIN {r1, r2, ..., rn}被定义为：如果 n=0，那么结果为 TABLE_DEE；如果 n=1，那么结果为 r1；否则，从集合 r1，r2，…，rn 中选择任意两个不同的关系并用二元情况下的连接结果替代它们，如此重复直到整个集合只剩下一个关系 r 为止，而这就是最后的结果。

- **MATCHING（匹配）**：表达式 r1 MATCHING r2 将返回一个关系等于表达式 r1 JOIN r2 所返回的关系，并将结果投影回 r1 的属性上。

- **MINUS（求差异）**：令关系 r1 和 r2 拥有相同的关系头 H。那么表达式 r1 MINUS r2 将返回一个关系拥有关系头 H，而关系体则由数组 t 组成，其中 t 出现在 r1 中，但不出现在 r2 中。

- **NOT MATCHING（去匹配）**：表达式 r1 NOT MATCHING r2 将返回一个关系等于表达式 r1 MINUS (r1 MATCHING r2)所返回的关系。

- **投影**：令关系 r 拥有关系头 H，并令 X 为 H 的子集，同时令 A1，A2，…，An 为所有 X 中的属性名称。那么在 X 上对 r 的投影 r{A1, A2, ..., An}则为这样一个关系：关系头为 X，关系体由数组 x 组成，使得在 r 中存在这样的数组 t 其值为 X 的值 x。

- **RENAME（重命名）**：令关系 r 拥有属性 A，而没有属性 B。那么表达式 r RENAME {A AS B}将返回一个关系，其关系头中除了属性 A 被重命名为 B 之外，其他与 r 的关系头完全一样。而其关系体中，除了所有数组对 A 的引用都被替换为对 B 的引用之外，其他与 r 的关系体完全一样。注意：本书的正文中并没有讨论过通过重命名操作来进行更新的问题，不过这是因为相应的更新规则实在是太显而易见了。

- **限制**：令 r 为一个关系，并令 bx 为一个布尔表达式。那么表达式 WHERE bx 将会返回一个关系，它拥有与 r 相同的关系头，而关系体则由 r 上所有对 bx 求值为 TURE 的数组组成。注意：技术上来说，bx 应当是限制条件，也就是说（a）所有的属性引用都是对 r 的属性引用，且（b）没有任何关系变量引用。然而，Tutorial D（与 SQL 类似）允许布尔表达式使用 WHERE 语句，这比一个单纯的限制条件要求更为宽泛。

- **SUMMARIZE（概览）**：1.（BY）令关系 r 拥有属性 A1，A2，…，An（以此类推），但是没有属性 B。那么表达式 SUMMARIZE r BY {A1, A2, ..., An} : {B := summary}将会返回一个关系，其关系头为{A1, A2, ..., An, B}，而关系体由数组 t 组成，其中数组 t 等于某些 r 上的数组在 A1，A2，…，An 上的投影，并且由值 b 来展开 B。这个值 b 是经由对 r 中所有满足下列条件的数组进行“summary”求值计算得来的，数组条件为：对属性 A1，A2，…，An 具有与 t 相同的值。2.（PER）令关系 r1 和 r2 存在这种关系，即 r2 的关系头是 r1 关系头的某个子集。令 r2 仅拥有属性 A1，A2，…，An，此外无他（尤其是没有属性 B）。那么表达式 SUMMARIZE r1 PER (r2) : {B := summary}将会返回一个关系，其关系头为{A1, A2, ..., An, B}，而关系体由所有数组 t 组成，其中 t 是 r2 的数组，并由值 b 对属性 B 进行展开。这个值 b 是经由对 r1 中所有满足下列条件的数组进行“summary”求值计算得来的，数组条件为对属性 A1，A2，…，An 具有与 t 相同的值。

- **TIMES（累加）**：1. （二元情况下）令关系 r1 和 r2 没有任何重名的属性。那么表达式 r1 TIMES r2 将会返回一个关系，其关系头为 r1 和 r2 关系头的并集，而关系体则由数组 t 组成，其中 t 为一个 r1 的数组和一个 r2 的数组的并集。2.（多元情况下）令关系 r1，r2，…，rn（n ⩾ 0）中任意两个关系之间没有任何重名的属性。那么表达式 TIMES {r1, r2, ...,rn}将会返回一个关系，其关系头为 r1，r2，…，rn 各关系头的并集，而关系体由数组 t 组成，其中 t 为 r1 的一个数组，r2 的一个数组……rn 的一个数组的并集。注意：TIMES 其实只是 JOIN 的一种特殊情况。

- **UNGROUP（去组合）**：令关系 s 拥有类型为 RELATION {Y1, Y2, ..., Yn}的属性 YR，并令 X={X1, X2, ..., Xm}为 s 中除 YR 外的所有属性；同时，令 Xi 不与任何 Yj 重名（0 ≤ i ≤ m, 0 ≤ j ≤ n）。那么表达式 s UNGROUP (YR)将会返回一个关系 r。r 的关系头为{X1, X2, ..., Xm, Y1, Y2, ..., Yn}。而 r 的关系体定义如下。令 z 为一个关系，拥有关系头{X1, X2, ..., Xm, YT}，其中 YT 的类型为 TUPLE {Y1, Y2, ..., Yn}，而关系体定义如下：对于每一个数组 s、z 都包含一个类型为{X, YT}的数组的集合，我们设这类数组为 t，那么每个数组 t 是数组 s 中对应 YR 的值，而每个这样的数组 t 中都包含一个与数组 s 中 X 值相等的 X 值，以及一个与数组 s 中 YR 值相等的 YT 值。令 z 不包含任何其他的数组，那么，r 为 z UNWRAP (YT)的结果。

- **UNION（合并）**：1.（二元情况下）令关系 r1 和 r2 拥有相同的关系头 H。那么表达式 r1 UNION r2 将会返回一个关系，其关系头也为 H，而关系体则由数组 t 组成，其中 t 至少在 r1 或 r2 二者之一中出现。2. （多元情况下）令关系 r1, r2, ⋯, rn（n ≥ 0）都拥有相同的关系头 H。那么表达式 UNION {r1,r2,...,rn}将会返回一个关系，其关系头为 H，而关系体则由数组 t 组成，其中 t 至少在 r1，r2，⋯，rn 其中之一中出现。注意：如果 n=0，那么（a）虽然在这里没有展示出来，但是需要在相应的关系头中详细指定某些语义的功能，（b）结果是具有相应关系类型的空集关系。

- **UNWRAP（去包裹）**：令关系 s 拥有一个类型为 TUPLE {Y1, Y2, ..., Yn}的属性 YT，并令 X = {X1, X2, ..., Xm}为 s 中除 YT 外的所有属性，令 Xi 不与任何 Yj 重名（0 ≤ i ≤ m, 0 ≤ j ≤ n）。那么表达式 s UNWRAP (YT)将会返回一个关系 r。r 的关系头为{X1, X2, ..., Xm, Y1, Y2, ..., Yn}。而 r 的关系体则包含一个数组与 s 中每个数组相对应，此外没有其他数组。令 s 中的数组 t 拥有 X 的值 x 以及 YT 赋值的数组包含 Y1 的值 y1，Y2 的值 y2⋯⋯以及 Yn 的值 yn，那么相应的 r 中的数组拥有 X 的值 x，Y1 的值 y1，Y2 的值 y2⋯⋯以及 Yn 的值 yn。

- **WRAP（包裹）**：令关系 r 的关系头分为两个子集，X = {X1, X2, ..., Xm}与 Y = {Y1, Y2 ..., Yn}，同时，令 YT 为一个属性，并不与 X 中任何属性重名。那么表达式 r WRAP ({Y1, Y2, ..., Yn} AS YT)将会返回一个关系 s。s 的关系头为{X1, X2, ..., Xm, YT}，其中 YT 的类型为 TUPLE {Y1, Y2, ..., Yn}。而 s 的关系体包含一个数组与每一个 r 中的数组相对应，此外没有其他数组。令 r 中的数组 t 拥有 X 的值 x，Y1 的值 y1，Y2 的值 y2⋯⋯以及 Yn 的值 yn，那么相应的 s 中的数组拥有 X 的值 x，以及 YT 赋值的数组包含 Y1 的值 y1，Y2 的值 y2⋯⋯

以及 Yn 的值 yn。

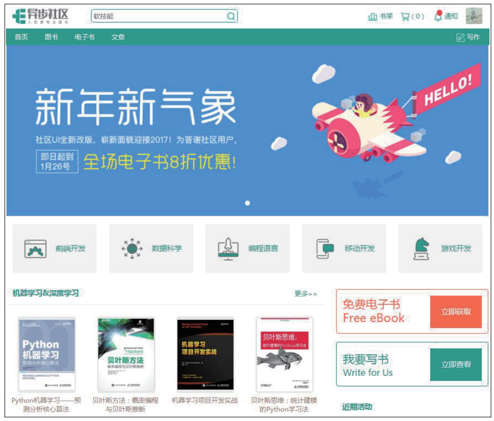
> 上述的运算符中，数组定义类似的为下面几个：EXTEND、投影、RENAME、UNION、UNWRAP，以及 WRAP（仅有这几个）。

最后，为了内容的完整性，下面我将给出“映像关系”概念的定义，这个概念我在第 8、12 以及 13 章中有提到过。

- **映像关系**：令关系 r1 和 r2 拥有具备相同名字和相同类型的属性；令 t1 是 r1 的一个数组；令 t2 是 r2 的一个数组，其所拥有的 r1 与 r2 共有属性的值，与 t1 中的一样；令关系 r3 为 r2 的限制，包含所有的数组 t2，并且没有其他数组；并令关系 r4 为 r3 在除 r1 与 r2 共有属性外所有属性上的投影。那么 r4 即为对应 t1 的（关于 r2 的）“映像关系”。注意：映像关系的调用只有在数组 t1 被完好定义的前提下才能进行。关于这个问题的更多内容，请大家参看《SQL and Relational Theory》。

# 欢迎来到异步社区！

## 异步社区的来历




异步社区（www.epubit.com.cn）是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。

异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD 按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

## 社区里都有什么?

### 购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

### 下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

### 与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题 可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

## 灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户账户中的积分可以用于购书优惠。100 积分 =1 元，购买图书时，在 

里填入可使用的积分数值，即可扣减相应金额。

## 特 别 优 惠

购买本书的读者专享**异步社区购书优惠券**。

使用方法：注册成为社区用户，在下单购书时输入 57AWG 使用优惠码，然后点击“使用优惠码”，即可享受电子书 8 折优惠（本优惠券只可使用一次）。

### 纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。



## 社区里还可以做什么？

### 提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得 100 积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

### 写作

社区提供基于 Markdown 的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

### 会议活动早知道

您可以掌握 IT 圈的技术会议资讯，更有机会免费获赠大会门票。

## 加入异步

扫描任意二维码都能找到我们：

| 异步社区 | 微信服务号 | 微信订阅号 | 官方微博 | QQ 群：436746675 |
|---|---|---|---|---|

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@ 人邮异步社区，@ 人民邮电出版社 - 信息技术分社

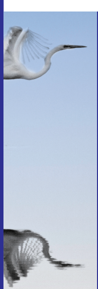
投稿 & 咨询：contact@epubit.com.cn

Databases

# 视图更新与关系数据库理论

视图就是“虚拟的表”。这意味着它们应该和“真”表或者基表一样，可以更新。实际上，不论是从实用角度还是从理论角度，视图的可更新性都是非常值得讨论的主题，甚至是关键性的技术。但是，视图更新却总是被视为一个有争议性的话题。

本书通过展示视图如何像基表一样，“总是”可以更新的（只要不违反任何一致性约束），进而完全颠覆传统观点。更宏观一点来说，本书展示的是更新应该“总是”可以进行的，无论目标对象是基表还是视图。本书所阐释的体系100%与关系模型一致，但是和今天SQL产品中的更新稍有不同。



通过本书，你将能够：

- 帮助数据库产品在未来进行改进；
- 在没有产品支持的情况下实现“自助式”实施；
- 了解谓词和约束的重要地位；
- 展示关系型产品到底应该如何运转。

任何与关系模型、关系技术或者数据库系统相关的专业人员，都可以从本书中获益。

## 作者简介

**Chris Date**是著名的数据库行业专家，善于用最简明易懂的方式解释复杂的技术问题。他于2012年出版了两本新书《SQL and Relational Theory，2nd Edition》和《Database Design and Relational Theory》，本书是它们的姊妹篇。



封面设计：Randy Comer，张健

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

分类建议：计算机/数据库
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-43546-0